TABLE OF CONTENTS

page

1.0 Introduction 1-1

2.0 The NABU Network 1-6

3.0 The NABU personal computer 1-9

3.1 Memory Organization 1-9

3.2 The TMS9918A Video Display Processor 1-10
3.2.1 Registers 1-13
3.2.2 Text Mode 1-15
3.2.3 Graphic 1 Mode 1-15
3.2.4 Graphic 2 Mode 1-18
3.2.5 Multicolour Mode 1-18
3.2.6 Sprites 1-19
3.2.7 VRAM table addresses 1-22
3.2.8 Graphics One Example 1-23

3.3 The AY-3-8910 Programmable Sound Generator 1-25

4.0 Internal Operating Software 2-1

4.1 Conventions Used by the IOS 2-2
4.1.1 Stack Operation and Requirements 2-4

4.2 Introduction to DOS 2-9
4.2.1 Segment Handling Routines 2-10
4.2.1.1 Introduction 2-10
4.2.1.2 Segment Control and Status Block 2-10
4.2.1.3 DOS Interface 2-13
4.2.1.4 Segment Headers 2-16
4.2.1.5 Examples 2-19

4.2.2 Directory Routines 2-23
4.2.2.1 Introduction 2-23
4.2.2.2 Format of Directory 2-23
4.2.2.3 Accessing the Directory 2-25

4.2.3 Interrupt Structure and Tasking Support 2-31
4.2.3.1 Introduction 2-31
4.2.3.2 Critical Regions 2-32
4.2.3.3 User Task Attachment Routines 2-34
4.2.3.3.1 Attaching Tasks to the Clock 2-34
4.2.3.3.2 Keyboard User Tasks 2-39
4.2.3.3.3 Expansion Slots 2-41

4.2.4 Human Input Devices 2-43
4.2.4.1 Introduction 2-43
4.2.4.2 Special Key Operation 2-43
4.2.4.3 Obtaining Data From the Keyboard 2-45
4.2.4.4 Sym Table Operation 2-47

4.2.5 Video Screen Device Driver 2-49

4.2.6 Printer Output 2-51

4.2.7 I/O router 2-52
4.2.7.1 Physical Device Identification 2-52
4.2.7.2 Logical Device Identification 2-53
4.2.7.3 I/O Routing Entry Point 2-53

4.3 Basic Operating Software 3-1

5.0 Extended IOS (XIOS) 4-1

5.1 Introduction 4-1

5.2 Extended IOS Module Handler 4-1
5.2.1 Memory Structure for Loaded XIOS Modules 4-2
5.2.2 Loading XIOS Modules 4-3
5.2.3 Unloading XIOS Modules 4-5
5.2.4 Resolving References in XIOS Modules 4-6

5.3 Disk System 5-1

5.3.1 Introduction 5-1

5.4 Multi-Window Screen Driver 6-1

5.4.1 Introduction 6-1
5.4.2 Operational Requirements 6-1
5.4.3 Module Specific Error Codes 6-1
5.4.4 Module Initialization 6-1
5.4.5 Module De-Initialization 6-2
5.4.6 DOS Call Interface 6-2
5.4.7 BOS Call Interface 6-7

5.5 80 Column Screen Driver 7-1

5.5.1 Introduction 7-1
5.5.2 Operational Requirements 7-1
5.5.3 Module Specific Error Codes 7-1
5.5.4 Module Initialization 7-1
5.5.5 Module De-Initialization 7-2
5.5.6 DOS Call Interface 7-2
5.5.6.1 Input Status from Video Screen Window 7-2
5.5.6.2 Output Data to Video Screen Window 7-3

5.6 CP/M Compatible Logical Device Drivers 8-1

5.6.1 Introduction 8-1
5.6.2 Operational Requirements 8-1
5.6.3 Module Specific Error Codes 8-1
5.6.4 Module Initialization 8-1
5.6.5 Module De-Initialization 8-1
5.6.6 DOS Call Interface 8-2

APPENDIX A
A.0 GLOSSARY 9-1

APPENDIX B
B.0 DOS and BOS number - Function Cross Reference 9-3

APPENDIX C
C.0 Sample Program and Documentation 9-6

THIS PAGE LEFT INTENTIONALLY BLANK

# 1.0 INTRODUCTION

This document is intended to provide the application programmer the necessary information and reference material to write application programs for the NABU personal computer. Complete programming information on the internal operating software (IOS) as well as programming information of the Video display processor and the programmable sound generator are included.

One of the aims of this manual was to collect all the information that was previously found in several documents into just one. Although this has yielded a document of some 200 pages, each section discusses a single concept related to the programming environment at NABU. Therefore the programmer need only investigate the portions of interest and not have to read the entire manual.

In order to put the IOS into perspective, we include here a section from the IOS Specification which spells out the general functional requirements of IOS. This will enable you to judge what to expect from the Internal Operating System.

## DESIGN REQUIREMENTS

### Overview

This design specification defines the Internal Operating Software (IOS) for the NABU Personal Computer (NPC), a low-cost, expandable personal computer. It is unique because it is capable of communicating on one-way, hybrid and two-way cable systems and telephone networks, as well as operating in a stand-alone mode, depending on which options are selected. When used in association with a CATV network the NABU P.C.'s prime function is to run software downline loaded from the cable head-end.

A versatile set of internal operating system and device handling software is required for the NABU P.C. to run applications software under control of a user. For definition and development purposes this software, collectively referred to as the Internal Operating Software (IOS) consists of:

- Applications program interfaces to IOS facilities
- All physical device control and I/O handlers
- Basic task controlling and interrupt handling software
- Communications Software

The internal operating software does NOT include:

- o Human Interface for Selection of Applications Programs
- o Any ROM Software in the NPC
- o Programming languages (eg. BASIC is not part of the operating system.)
- o Monitors (ie. examine and change memory, etc., etc. ,etc)
- o High-level (user oriented) utilities

## Operating Environment

The IOS must interact with four other functional components of the NABU P.C.. These are:

- o The Basic NABU P.C. hardware
- o Optional hardware and peripheral devices
- o Communications with external systems, including the keyboard and NABU Adaptor (NA)
- o Applications Software

It is the requirements and functions of these components which essentially define the requirements for the IOS.

## Internal Operating Software Requirements

The fundamental requirement of the Internal Operating Software is to create an environment which supports the loading and execution of applications programs in a simple, efficient manner. The NPC hardware, its peripherals, communications and the IOS are really just necessary evils required to present content to an NPC user. The IOS provides a stable interface which allows applications access to the other NPC components while hiding the messy details of the hardware configuration and communications protocols, which are really of no interest to applications programs.

## IOS Flexibility

In order to be as flexible as possible, the IOS resides completely in RAM. A separate program, the MAIN MENU program, is loaded in along with the IOS when the NABU P.C. is "booted". The MAIN MENU performs all human interface functions required to load in an application.

A number of expansion options will be offered for the NPC. These options may include: standalone operation through use of ROM readers and/or floppy disks, additional communications options though the use of telephone dialers, two-way cable modems and other devices, and the support of various other peripherals via an I/O expansion bus. The IOS must be able to operate in a configuration independent manner. This implies:

- The IOS must be able to sense the NPC configuration when "Booted"
- The IOS should protect the applications from becoming "configuration-dependent"
- Standard I/O handling procedures and I/O routing must be included in the IOS
- The IOS may be required to operate using different types of primary storage devices.

## Applications Interfacing

As was mentioned earlier the NPC and IOS exist to run applications. In this sense applications software is the highest level of software and it is in control of the IOS. Different applications have different requirements. Animated video games and other applications which require rich active human interfaces will require fast, efficient, unadorned access to NPC devices. At the other end of the scale are many of the computation type applications which are willing to sacrifice speed for I/O independence and ease of use. Other software such as a screen-oriented word processor lies between the two extremes of support.

This implies:

- Applications must have as much control as possible over the IOS
- Applications should be able to access IOS features at a number of different levels
- IOS support should be designed to fit applications requirements and not vice versa

## Real Time Requirements

Unlike many other microcomputer operating environments, the NPC will have time-critical tasks. The most obvious of these is communications on the CATV network. However many of the applications planned for the NPC have real-time components.

This implies:

- The lower layers of the IOS must be as time-efficient as possible
- Interrupts must be well supported in the IOS
- Applications software has as much control as possible over the enabling of interrupts and the complexity of interrupt handling
- Some simple tasking constructs should be provided
- Attachment of applications supplied code to interrupt handlers should be supported where possible
- Real-time counters (60Hz rate) should be supported by the IOS

## Application Time-out Requirement

Due to the T.V. screen being used for the basic output device, if no keyboard input is received for long periods of time (approx. 20 to 30 minutes), the T.V. screen will go blank (to prevent burning of the TV screen). This assumes that the clock interrupt is running, inorder to do the timing. The program execution must continue even though nothing is being displayed. When any key on the keyboard is activated, the T.V. screen will return back to its normal display. The keystroke which re-activates the screen is not passed on to the application program. {This time-out will also be active if the NPC is in the "PAUSED" mode.} ~~The only exception to time-out requirement is the case where the N.P.C. is in a "halt" mode because the PAUSE key has been activated. The PAUSE function causes the IOS to execute in a very tight loop, until PAUSE function is deactivated. This tight loop scans the keyboard for the activation of the PAUSE, TV/NABU, and SYM keys.~~

## Size Requirements

The total size of the IOS Kernel should not exceed 10K bytes and shall be kept to a minimum. In order to accommodate all the different IOS functions, the IOS will be divided into two sections. The first section will be called the Kernel. This will form a "bare bones" type IOS. The remainder of the IOS will form the second section which is called the Extended IOS (XIOS). As applications require functions which are only found in the XIOS, the application will be able to load in the necessary sections (modules) of the XIOS, and then use the functions. When the functions are no longer necessary, the XIOS module can be deleted, thus freeing up memory space.

## Internal Operating Software Structure

The Internal Operating Software is divided into three functionally separate components. These components are: the I/O handlers, the Basic Operating Software (BOS), and the Downloadable Operating Software (DOS).

### I/O Handlers

These portions of the software contain the low-level controlling code to handle input and output devices. Each physical device has its own I/O handler. This software masks the detailed physical operation of peripheral devices so that the higher levels of the operating system may be peripheral device independent. I/O Handlers provide:

- Hardware Dependent Device Control Code
- Interrupt Handling
- Initialization Code
- Data Link Layer Communications Protocols

### Basic Operating Software (BOS)

This level of the operating system provides the key operating control software for the NABU P.C.. It interfaces to the I/O handlers, the Downloadable Operating Software and applications programs. The BOS provides:

- Functional Level I/O handling
- Calling of I/O handlers and device control code
- Interrupt and task handling control
- A Method of Linking Directly to each BOS Routine

### Downloadable Operating Software

This is the highest layer of the internal operating software. It interfaces to the BOS and applications programs to provide:

- Common Entry Points for Applications
- I/O Routing
- Configuration Identification

## 2.0 THE NABU NETWORK

The NABU Network was formed on the idea of linking a microcomputer to the cable network. The union of these two technologies has paved the way for the introduction of a microcomputer complete with a large base of software into the homes of the population at large.

This section will describe the various links in the chain of this Network with a view to giving a broad understanding of the pathway followed by an application program from the cable company to the end user's RAM. Refer to the diagram for a pictorial representation of this data flow.

### The Head End

As the name suggests, this is the originating node in the Network. The Head End is actually a minicomputer and it is here that all the programs and data to be broadcast on the cable are found. The Head End minicomputer is constantly outputing the information in its database and it does so in a cyclic fashion - when all the information has been sent, the mini starts at the beginning and re-sends the database. This cyclic nature of the data flow enables one to envision the data as being written on the edge of a wheel which is read as it revolves.

Each application on the "wheel" is tagged with an identification number. This number becomes important at the other end of the NABU Network to select the proper user application.

The Head End is also responsible for the maintenance of this database. Any additions or deletions must be carefully dealt with in order to ensure the overall integrity of the information as these changes will alter the "diameter" of the "wheel".

### The RF Modulator

The information output by the Head End mini is of course digital in nature. Before this can be put onto the cable, the data signal must be modulated. The RF modulator will perform this function.

## The Combiner

Since there are other services on the cable (eg. TV, radio), there must be another piece of equipment that will merge the NABU programs with that information. The Combiner performs this task. The NABU information is now broadcast on a specific channel and sent into the cable for distribution.

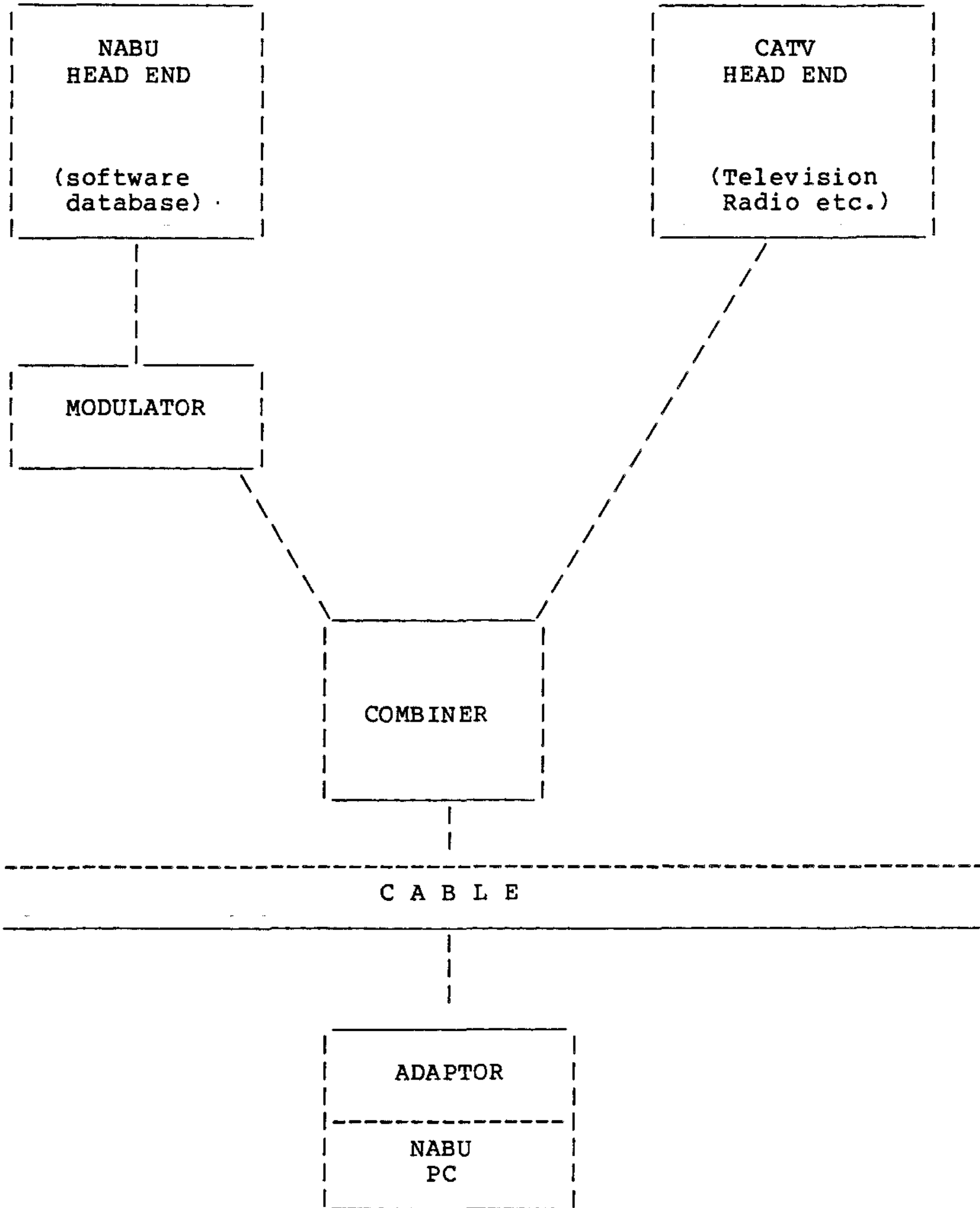
## The Adaptor

The Adaptor is a piece of hardware that acts as the interface between the cable coming into the home of the NABU user and the NABU Personal Computer.

Essentially, the Adaptor performs the reverse functions of the Combiner and the RF Modulator. It is tuned to listen to the NABU channel, de-modulate the signal and convert it into the digital data that the NABU PC can understand.

On the cable side, the Adaptor is only capable of listening to the information coming down the cable - it cannot send commands back to the Head End. However, on the PC side of the Adaptor there is two-way communication. The PC can tell the Adaptor what it wishes from the cable and the Adaptor can inform the PC when that data is available to be read.

Thus, when the user requests a particular application, the PC sends a Read command and the identification number of the application to the Adaptor. The Adaptor then "listens" to the cable until the appropriately identified data appears. The Adaptor fills its internal buffer and then informs the PC that the data is ready. The PC obtains the data from the Adaptor putting it into the appropriate location in the RAM of the PC.




- THE NABU NETWORK -

# 3.0 THE NABU PERSONAL COMPUTER

## INTRODUCTION

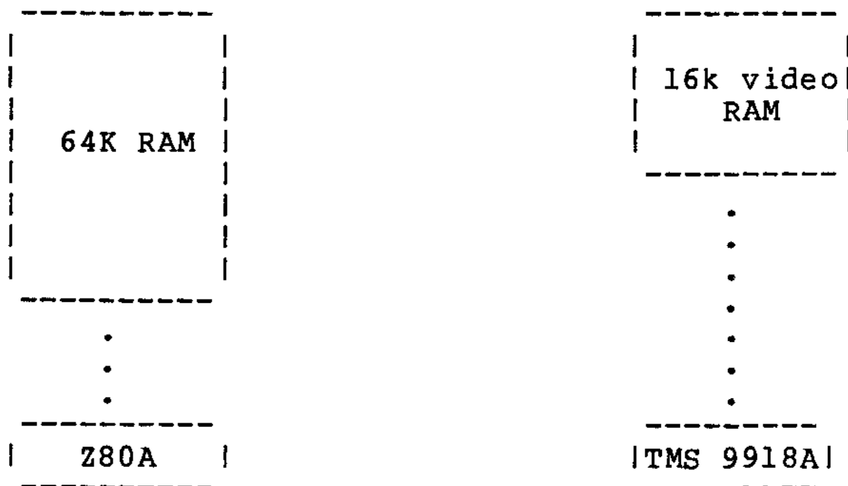
This section will provide the application programmer the necessary introduction and information to the hardware of the NABU Personal Computer.

## 3.1 MEMORY ORGANIZATION

The NABU Personal Computer is a 80 Kbyte machine. The 80K is partitioned as follows:

1) The primary memory is 64K in size. It is the only region where Z80 microprocessor code may be executed.

2) A 16K block of memory is dedicated for use by the TMS 9918A video display processor.




The above figure graphically describes the memory organization.

## 3.2 THE TMS 9918A VIDEO DISPLAY PROCESSOR

The TMS 9918A Video Display Processor (VDP) is responsible for all video display for the NABU Personal Computer (NPC). It provides for text, graphics and animation. Detailed knowledge of the control of the VDP is not required since all functions of the VDP are accessed through routines provided in the Internal Operating System (IOS) of the NPC. This section will outline the features of the VDP and the use of IOS routines to generate T.V. images for display on the NPC. Further information may be found in the Texas Instruments 9900 Data Manual (TMS9918A/TMS9928A/ TMS9929A Video Display Processors).

The VDP produces a T.V. image that can be envisioned as a series of display planes. Each plane has a display priority. An image on a plane of higher priority will overwrite an overlapping image on a lower priority plane. The display planes in order of lowest to highest priority are BACKDROP, PATTERN, and SPRITE. Sprites are special animation objects. The VDP provides 32 sprite planes, with sprite plane 1 having the highest priority.

The lowest priority plane is the BACKDROP, which consists of a single colour. It can be set to any one of 15 colours. The area covered by the backdrop plane is larger that the other planes, and can form a border for the pattern plane. With the T.V. displays commonly used with the NPC, the border effect is generally limited to the top and bottom of the screen, while the side borders are cropped by the T.V. overscan. The colour of the backdrop is determined by write-only register 7 of the VDP (see 3.2.1 REGISTERS).

The image displayed in the pattern plane is determined by the contents of 16K of Video RAM (VRAM) provided for the VDP. The contents of the PATTERN NAME TABLE (Name Table), PATTERN GENERATOR TABLE (Pattern Table), and COLOUR TABLE allocated in VRAM define the pattern plane image. The mode of the VDP determines the size and organization of the tables and hence the way in which VRAM is mapped to the screen. The VDP can operate in any one of four modes, Text, Graphics I, Graphics II, and Multicolour.

The images displayed in the sprite planes are defined in the SPRITE ATTRIBUTE TABLE and SPRITE PATTERN GENERATOR TABLE. These tables are also allocated in VRAM, and perform the sprite equivalents of pattern plane tables.

The VDP produces a screen image with an absolute resolution of 256 X 192 pixels. The VDP divides the pattern plane into blocks of pixels called patterns. In Text mode, the patterns are 6 X 8 pixels, yielding 40 text pattern per line. In Graphics modes the patterns are 8 X 8 pixels (32 patterns per line). There is a one byte entry in the Name Table for each pattern position on the screen. For example, in Graphics modes, the Name Table is 768 bytes long (32 patterns per row X 24 rows of patterns). In Text mode, the Name Table is 960 bytes long (40 X 24). There is a one-to-one mapping of entries in the Name Table and screen pattern positions (see Figure 1 for example). The screen origin is defined as the top left corner.

```
+---+---+ - - - - - - - +---+---+
|  0|  1|               | 30| 31|
|---+---+ - - - - - - - +---+---|
| 32| 33|               | 62| 63|
|---+---+ - - - - - - - +---+---|
.   .   .               .   .   .
.   .   .               .   .   .
|---+---+ - - - - - - - +---+---+
|704|705|               |734|735|
|---+---+ - - - - - - - +---+---|
|736|737|               |766|767|
+---+---+ - - - - - - - +---+---+
```

Fig. 1. Graphics I Name Table Mapping
The figure illustrates the pattern positions on a T.V screen with the VDP in Graphics I mode. The number associated with each position maps to the entry (offset) within the Name Table. The 0th entry in the Name Table maps to the pattern position occupying the top left corner of the screen.

The Pattern Table determines which pixels will be turned on within a pattern. Each entry in the Pattern Table is eight bytes long. The first byte of an entry defines the pixel arrangement of the top row of a pattern, the second byte the second row and so on. A '1' bit specifies a pixel that is on and a '0' bit specifies a pixel that is off. The offset of an entry into the Pattern Table (i.e. the entry number) forms the 'name' of the pattern. A pattern can be displayed on the screen in any pattern position by writing its name (offset) to the appropriate entry in the Name Table. The number of patterns available in the Pattern Table depends on the mode of the VDP.

The VDP is capable of producing fifteen colours plus transparent. The Colour Table determines the colours of the pixels defined in the Pattern Table. The high order nibble of a byte in the Colour Table defines the colour of the '1' bits in the associated byte of the Pattern Table. The low order nibble defines the colour of the '0' bits. The resolution of the mapping from Colour Table to Pattern Table is dependent on the mode of the VDP. The colours associated with each 4 bit nibble are shown in Table 1.

The base addresses of the VRAM tables are derived from the values contained in the VDP's write-only registers, and are subject to restrictions dependent on the mode of the VDP. The base addresses are defined by calling the specific IOS routine for that table, which will set the correct bits in the appropriate VDP register. This process does not require a knowledge of the register addressing scheme.

| HEX VALUE | COLOUR |
|---|---|
| 0 | Transparent |
| 1 | Black |
| 2 | Medium Green |
| 3 | Light Green |
| 4 | Dark Blue |
| 5 | Light Blue |
| 6 | Dark Red |
| 7 | Cyan |
| 8 | Medium Red |
| 9 | Light Red |
| A | Dark Yellow |
| B | Light Yellow |
| C | Dark Green |
| D | Magenta |
| E | Gray |
| F | White |

Table 1. Colour Assignments
The 4 bit hex values in the first column
produce the colour in the second column

### 3.2.1 REGISTERS

The VDP is equipped with eight write-only registers and a single read-only status register. The write-only registers are used to define the mode of the VDP, table addresses in VRAM, and the backdrop colour. All access to these registers is by way of calls to routines in the IOS. Descriptions of these routines can be found in the section on BOS calls.

The write-only registers may be loaded with the IOS routine VREGWR. Specialized routines are provided for specifying VRAM table addresses. In the NPC environment a RAM image of the write-only registers is maintained, allowing examination of register contents. The registers may be 'read' by calling VREGRD, or with specialized routines (see IOS document).

REGISTER 0
REGISTER 1

These two registers contain VDP option control bits. In practise, they are not written to directly with VREGWR, but rather are accessed through specialized routines. VSETXT is called to set the appropriate bits to place the VDP in TEXT mode. Other routines are VSETG1 (Graphics I) and VSETG2 (Graphics II).

The VDP also has a 'vertical blanking' option (the video screen is "blacked out") which is selected in register 1. The screen may be blanked with no effect on VRAM by calling the IOS routine VBLKON. The screen is restored with VBLKOFF. Other bits in register 1 determine the size and magnification of sprites (see 3.2.6 SPRITES).

REGISTER 2

Register 2 defines the base address of the Name Table. The address is set by calling VNAMEST.

REGISTER 3

Register 3 defines the base address of the Colour Table. The address is set by calling VCOLRST.

REGISTER 4

Register 4 defines the base address of the Pattern Generator Table. The address is set by calling VPTRNST.

REGISTER 5

Register 5 defines the base address of the Sprite Attribute Table. The address is set by calling VATRIST.

REGISTER 6

Register 6 defines the base address of the Sprite Pattern Generator Table. The address is set by calling VSPRIST.

REGISTER 7

The high order 4 bits of register 7 define the colour code of '1' pixels in Text mode. The low order bits define the colour code for '0' pixels in Text mode and the backdrop colour in all modes. Register 7 is loaded by calling VREGWR.

STATUS REGISTER

The status register contains the following flags.

F - The Interrupt Flag is set at the end of the raster scan of the last line of the display. It is reset to 0 after the VDP Status Register is read or the VDP is reset.

C - The Coincidence Flag flag is set whenever two sprites have '1' bits at the same screen location. (see 3.2.6 SPRITES).

5S - The Fifth Sprite Flag is set whenever more than four sprites are displayed on the same horizontal line. The number of the fifth sprite is also loaded into the VDP Status Register. (see 3.2.6 SPRITES).

### 3.2.2 TEXT MODE

As is implied by the name, Text mode is primarily for textual applications. The Name Table and Pattern Table are used to define the appearance of the screen. The Colour Table is not used. Patterns are 6 X 8 pixels, which allows for an increase to 40 characters per line. The Name Table is 960 (40 X 24) bytes. The Pattern Table contains the library of text patterns to be displayed. It is 2048 bytes long, consisting of 256 eight byte entries. Since each text position is only 6 pixels wide, the two least significant bits of each row of the pattern are ignored. There can only be two colours for the entire screen, one colour for all of the '1' bits, and a second colour for all of the '0' bits. The colours are defined in VDP register 7 (see 3.2.1 REGISTERS).

Typically, text patterns are loaded into the Pattern Table, such that the entry number corresponds to the ASCII code for the letter. For example, the ASCII code for the letter 'A' is 65 (decimal). With the eight byte pattern for the letter 'A' occupying pattern number 65 in the Pattern Table, the letter can be written to screen pattern position 3 by writing 65 to the third entry in the Name Table (Figure 2).

Text mode allows for 40 characters per line on a T.V display. However, because of T.V. overscan, characters should not be written to columns 0,1,38 or 39. This effectively reduces the display to 36 characters per line.

### 3.2.3 GRAPHICS I MODE

The VRAM tables that are used to generate the screen image for Graphics I mode are the PATTERN NAME TABLE (Name Table), PATTERN GENERATOR TABLE (Pattern Table) and COLOUR TABLE. The Name Table determines the screen position for a pattern. The Pattern Table determines which pixels within a pattern will be turned on. The Colour Table determines the colour of a pixel.

The VDP divides the screen into 8 X 8 pixel patterns, meaning that the Name Table has 768 one byte entries. The Pattern Table contains a library of patterns that may be placed in any pattern position on the screen. The Pattern Table is 2048 bytes long, consisting of 256 eight byte entries. There is a maximum, therefore, of 256 unique patterns which may be displayed at any one time in Graphics I. The offset of the pattern within the Pattern Table forms the name of the pattern. To display a pattern at a specific position on the screen, the pattern name is written to the appropriate entry in the Name Table.

```
NAME TABLE

+---------+ <-- ENTRY 0
.         .
.         .
+---------+ <-- ENTRY 3
|  (65)   |
+---------+
.         .
.         .
+---------+ <-- ENTRY 95
|  (66)   |
+---------+
.         .
.         .
+---------+ <-- ENTRY 959
|         |
+---------+
```

```
PATTERN TABLE

+---------+ <-- ENTRY 0
.         .
.         .
+---------+ <-- ENTRY 65
|000000--|
|001000--|
|010100--|
|100010--|
|111110--|
|100010--|
|100010--|
|100010--|
+---------+ <-- ENTRY 66
|000000--|
|111100--|
|100010--|
|100010--|
|111100--|
|100010--|
|100010--|
|111100--|
+---------+ <-- ENTRY 67
.         .
.         .
+---------+ <-- ENTRY 255
.         .
+---------+
```

T.V. Display has 'A' (pattern 65) in screen position 3, and 'B' (pattern 66) in position 95.

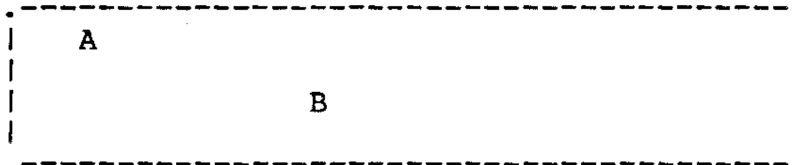



Fig. 2. Name and Pattern Table Mapping in Text Mode

The colours of pixels are specified in the Colour Table. The colour table contains 32 one byte entries. Each entry defines two colours, the high order nibble of each entry defines the colour of the '1' bits, and the low order nibble defines the colour of the '0' bits. The first entry in the Colour Table defines the colours for patterns 0 - 7, the second entry for patterns 8 - 15 and so on. This scheme imposes the following colour restrictions: 1) any one pattern can only display two colours and 2) changing the colours for one pattern implies a colour change for the seven other patterns within the colour group.

### 3.2.4 GRAPHICS II MODE

Graphics II mode is similar to Graphics I mode except that the Pattern and Colour Tables are longer.

The Pattern Table is expanded to 6144 bytes, allowing for 768 unique patterns, one for each pattern position on the screen. Since the one byte entries in the Name Table allow for a maximum of 256 unique entries, Graphics II segments the Name Table into three blocks of 256 names each such that the first block maps pattern names to the upper third of the screen. The second and third blocks map pattern names to the middle and lower thirds of the screen respectively. The Pattern Table is similarily segmented. Entries in the first third of the Name Table map to patterns in the first third (2048 bytes, 256 patterns) of the Pattern Table.

The Colour Table is also expanded to 6144 bytes. There are 768 eight byte entries. Thus, there is one eight byte entry in the Colour Table for each eight byte entry in the Pattern Table. The high order nibble of each byte defines the colour of the '1' bits in the corresponding byte of the Pattern Table. The colour of the '0' bits is defined by the low order nibble. Thus in Graphics II mode, two colours may be defined for each row (byte) of a pattern. The Colour Table is segmented into three equal parts in the same manner as the Pattern Table.

### 3.2.5 MULTICOLOUR MODE

The VRAM tables that need to be allocated for Multicolour mode are the Name Table and Pattern Tables. The Colour Table is not used, colours are derived from the Pattern Table. As Multicolour mode is rarely used, a complete description is not provided in this document. Further information may be found in the Texas Instruments 9900 Data Manual.

The pattern plane is divided into blocks of 4 X 4 pixels (64 X 48 blocks). The colour of each block can be any one of the fifteen video display colours plus transparent. The backdrop and sprite planes are active.

The Name Table consists of 768 one byte entries. The name points to an 8 byte segment of VRAM in the Pattern Generator Table. The colour to be displayed is determined by the information contained in the Pattern Table.

### 3.2.6 SPRITES

Sprites are special animation patterns. Up to 32 sprites are available, one for each of the sprite planes. Sprites may be used in Multicolour and Graphics modes, but not in Text mode. Each of the sprites can cover an 8 X 8, 16 X 16, or 32 X 32 pixel area on its plane. Any part of the plane not covered by the sprite is automatically transparent. All or part of each sprite can also be transparent. The highest priority sprite is 0, the lowest priority is sprite 31. All sprites are of higher priority than the pattern and backdrop planes. The location of a sprite is defined by the top leftcorner of the sprite pattern. The sprite can be easily moved pixel-by-pixel by redefining the sprite origin (IOS call SPMOVE).

The Sprite Attribute Table and the Sprite Generator Table are allocated in VRAM. These tables are the sprite equivalents of the Pattern Name Table and Pattern Generator Tables. Each entry in the Attribute Table is four bytes long, with one entry for each of the 32 available sprites. The first byte of an entry defines the vertical position of the sprite from the top of the screen in pixels. Values between -32 and 0 allow a sprite to bleed in from the top edge of the backdrop. A value of -1 causes the sprite to be positioned at the top of the screen, touching the backdrop area. The second byte defines the horizontal position of the sprite from the left edge of the display. A value of 0 positions the sprite against the left edge of the backdrop. The third byte defines the name of the sprite. This name maps to the Sprite Generator Table in the same way patterns are mapped from the Name Table to the Pattern Table. The low order four bits of the fourth byte contain the colour code for the '1' pixels of the sprite ('0' pixels are transparent). The most significant bit of the fourth byte is the Early Clock Bit. When set to '1', the position of the sprite is shifted to the left by 32 pixels, allowing the sprite to bleed in from the left edge of the display.

The Sprite Generator Table has up to 256 eight byte entries, for a maximum of 2048 bytes long, and is equivalent in function to the Pattern Generator Table. The IOS routine VSETSP is used to set the size and magnification of the sprites. Sprite size can be either 8 X 8 or 16 X 16 pixels. With 8 X 8 pixel sprites, the Generator Table uses eight bytes to define the sprite. When 16 X 16 sprites are used, the Generator Table requires 32 bytes. A 16 X 16 sprite is effectively divided in to four equal quadrants, with the bytes in the Generator Table being mapped to the screen as shown in Figure 3. The sprites can also be magnified one or two times. With a magnifaction factor of two, each bit in the Generator Table is mapped into 2 X 2 pixels on the screen display.

There is a limit of four sprites on any horizontal line. If more sprites are positioned to the same line, the four highest priority sprites are displayed normally. The fifth and subsequent sprites are not displayed on that line. The fifth sprite flag is set and the number of the fifth sprite is loaded into the VDP Status Register.

A value of D0 (hex) in the vertical position field of an entry in the Sprite Generator Table terminates sprite processing. This allows programmers to blank part or all of the sprites. The IOS routine SPMARK will write D0 (hex) to any sprite, and marks the end of the active sprites in the Attribute Table.

Whenever two active sprites have '1' bits at the same screen location, the coincidence flag in the VDP Status register is set.

```
byte
     +---------+
 0   |         |
     .    I    .
     .         .
 7   |         |
     +---------+
 8   |         |
     .   II    .
     .         .
 0F  |         |
     +---------+
 10  |         |
     .   III   .
     .         .
 17  |         |
     +---------+
 18  |         |
     .   IV    .
     .         .
 1F  |         |
     +---------+

  SPRITE GENERATOR
    TABLE ENTRY
```

```
+---------+---------+
|    I    |   III   |
|         |         |
+---------+---------+
|   II    |   IV    |
|         |         |
+---------+---------+
 SCREEN DISPLAY OF
 SPRITE PATTERN
```

Fig. 3 Sprite Generator Table Mapping for 16 X 16 (1X magification) sprites.

### 3.2.7 VRAM TABLE ADDRESSES

There are certain restrictions on where tables may be located in VRAM, dependent on the mode of the VDP. For example, in Text mode, the Pattern Table is 2048 bytes long, and must start on a 2 Kilobyte boundary in VRAM.

| VDP MODE | TABLE | LENGTH (max) | VRAM BOUNDARY |
|---|---|---|---|
| Text | Name | 960 | 1K |
| | Pattern | 2048 | 2K |
| Graphics I | Name | 768 | 1K |
| | Pattern | 2048 | 2K |
| | Colour | 32 | 64-byte |
| | Sprite Attribute | 128 | 128-byte |
| | Sprite Generator | 2048 | 2K |
| Graphics II | Name | 768 | 1K |
| | Pattern | 6144 | 8K |
| | Colour | 6144 | 8K |
| | Sprite Attribute | 128 | 128-byte |
| | Sprite Generator | 2048 | 2K |

The conventions for the NPC environment are that the Pattern Table always starts at VRAM address 0. The Name Table is placed at the next available boundary. In Graphics II mode, the Pattern Table is located at VRAM address 0, and the Colour Table at 8192.

Note that in Text and Graphics I modes several screens may be defined in VRAM with multiple Name Tables. Each Name Table starts on a 1K boundary. To display a particular screen, the address of the desired Name Table is written to VDP register 2 with the IOS routine VNAMEST. This is particularly useful for setting up several screens of text.

### 3.2.8 GRAPHICS I EXAMPLE

It is desired to place the following pattern at screen pattern position 255 (hex FF) in Graphics I mode using pattern number 8 in the Pattern Table. The pattern is to be black on a grey background.

```
*      *
 *    *
  *  *
   **          each * represents one pixel on the screen
   **
  *  *
 *    *
*      *
```

```
   PATTERN TABLE
entry 00  -->  +--------+
               |        |
               .        .
               .        .
entry 08  -->  +--------+     (BIT  MAP)
               |   81   |     (10000001)
               |   42   |     (01000010)
               |   24   |     (00100100)
               |   18   |     (00011000)
               |   18   |     (00011000)
               |   24   |     (00100100)
               |   42   |     (01000010)
               |   81   |     (10000001)
               +--------+
               .        .
               .        .
```

```
   NAME TABLE                                 COLOUR TABLE

entry 0  --> +--------+          entry 0  --> +--------+
             |        |                       |        |
             +--------+          entry 1  --> +--------+
             .        .                       |   10   |
             .        .                       +--------+
entry FF --> +--------+                       .        .
             |   08   |                       .        .
             +--------+
             .        .
             .        .
```

The '10' (hex) in the Colour Table sets the '1' bits in patterns 8 - 15 to black and the '0' bits to transparent. To set the screen to grey, VDP register 2 is set to '0E' (hex).

## 3.3 The Programmable Sound Generator

As previously mentioned, all sounds produced by the NPC are under the control of the AY-3-8910 programmable sound generator (PSG). This device uses 14 registers to generate a variety of complex sounds. The PSG has three individual channels to produce the sound effects.

Producing sounds using the audio generator may be divided into several sound generating blocks. They are:

1) tone generators
2) noise generator
3) amplitude control
4) envelope control

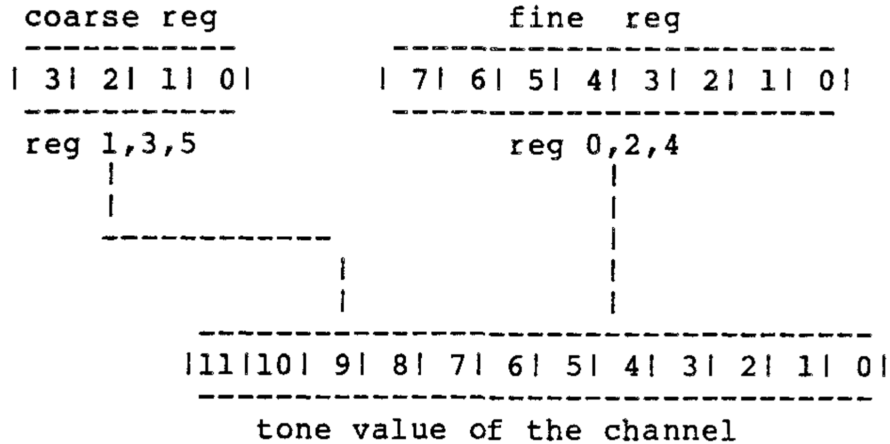
The registers of the PSG are used to enable/disable each of these blocks and to select the parameters of the channel in the PSG To read or write to the registers of the sound chip, the IOS BOS routines AUDWR and AUDRD MUST be used. See section 4.3

Register 0 and register 1 provide the period or frequency of the tone to be produced by channel A of the PSG. All 8 bits of register 0 is used but only the lower 4 bits of register 1 are used. This provides a tone frequency resolution of 12 bits with register 1 containing the most significant 4 bits and register 0 providing the remaining 8 least significant bits.

Register 2 and register 3 provide the period of frequency of the tone to be produced by channel B. Twelve bit resolution is provided with register 3 providing the most significant 4 bits.

Similarily Register 4 and register 5 provide the tone period for channel C with register 4 providing the most significant 4 bits of the twelve bits.

The following diagram should clarify the above information.




There are two formulae that relate output tone frequency to the value in the twelve bit register. They are:

$$f = \frac{223{,}750}{tp}$$

and

$$tp = 256ct + ft$$

Where
- f = the frequency of the sound to be generated
- tp= the tone period to be written to the registers
- ct= the coarse tune register (registers 1,3,5)
- ft= the fine tune register (registers 0,2,4)

*Register 6 provides the tone period of the noise to be* generated. It uses only the least significant 5 bits of register 6 and is the only register controlling the noise frequency.

Similarily, the freqency of the output tone may be related to the noise period by the following formula:

$$f = \frac{223750}{np}$$

Where f = the frequency of the noise to be generated
np = the noise period to be written to register 6

Register 7 enables and disables each of the three channels. Register 7 uses inverted logic therefore, a 1 indicates that the channel is disabled.

```
 channel        | C   B   A | C   B   A |
 function       |   NOISE   |   TONE    |
          -------------------------------
          | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  -register 7
          -------------------------------
```

Bits 7 and 6 are not used for generating sounds.

Register 8, 9 and 10 controls the amplitude for channels A, B, and C respectively as well as the envelope pattern.

If bit 4 is zero then the least significant 4 bits provide the amplitude (volume) of the channel's sound. This provides 16 levels of amplitude with 15 being the greatest and 0 producing no sound.

If bit 4 is 1 then envelopes are enabled and amplitude of each channel is determined by the envelope pattern as defined by the lower four bits of the register.

The remaining registers 11, 12 and 13 provide envelope control. There are two ways of controlling envelopes. First is to vary the frequency of the envelope using registers 11 and 12, the second way is to vary the shape and cycle pattern of the envelope.

The envelope period may be resolved to 16 bits by combining registers 11 and 12. Register 12 provides the most significant 8 bits and register 11 provides the least significant 8 bits. As before, 2 formulae may be used to relate the envelope period to the output envelope frequency. They are:

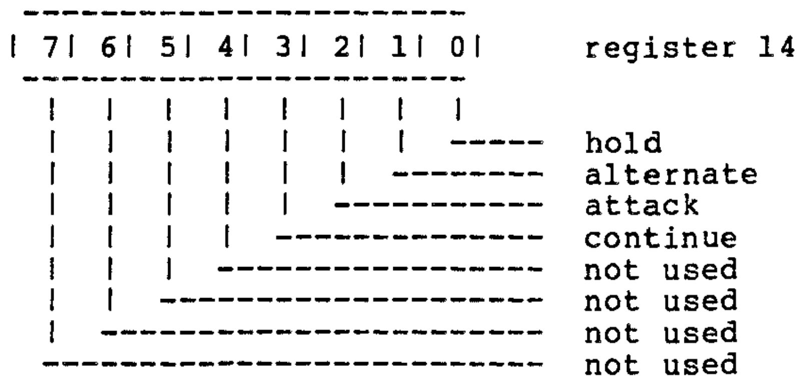
$$f = \frac{13984}{ep}$$

and

$$ep = 256ct + ft$$

Where
- f =the desired envelope frequency
- ep =the envelope period
- ct =the coarse tune register (reg 12)
- ft =the fine tuning register (reg 11)

Register 14 controls the envelope shape or cycle. Only the lower four bits of the register are used. Each bit has an individual function.




If hold is set to 1 the envelope is limited to one cycle and holds the current state of the envelope counter.

If alternate is set to 1 the envelope reverses the direction after each cycle.

If attack is set to 1, the envelope will count up.
If attack is set to 0, the envelope will count down.

If countinue is set to one the cycle pattern will be defined by the hold bit otherwise the envelope counter will reset to zero and then hold.

# 4.0 THE INTERNAL OPERATING SOFTWARE

The INTERNAL OPERATING SOFTWARE (IOS) is a versatile operating system used to run the application software. It provides a standard interface and sets of common routines to link the applications to the hardware of the NPC.

The IOS may be broken into 2 distinct portions for the application programmer. They are:

1) The Downloadable Operating Software (DOS)
2) The basic operating software (BOS)

## Downloadable Operating Software

This is the highest layer of the internal operating software. It interfaces to the BOS and applications programs to provide:

- o Configuration Identification
- o Functional Level I/O handling
- o Calling of I/O handlers and device control code
- o Interrupt and task handling control
- o Common Entry Points for Applications
- o I/O Routing

## Basic Operating Software

This level of the operating system provides the key operating control software for the NABU P.C.. It interfaces to the Downloadable Operating Software and applications programs. The BOS provides:

- o Functional Level I/O handling
- o Common entry points for applications.

## 4.1 CONVENTIONS USED BY THE INTERNAL OPERATING SOFTWARE

The IOS memory map structure is similar to that of Digital Research's CP/M operating system. Thus, any CP/M system may be used as a development system for the NABU P.C. Programs written in a high level language compatable with CP/M will run under the IOS. However there are differences between CP/M and IOS. Not all CP/M calls are implemented in the basic IOS and the stack requirements are different.

The memory map below IOS is layed out as follows:

```
                 ----------------------------
BASE to FFFFH|   reserved for  IOS          |
             |----------------------------|
    BASE-1   |   Applications Program       |
    0100 hex:|      Area + Stack(s)         |
             |----------------------------|
    00FF hex:|   Reserved Area for IOS      |
    000B hex:|                              |
             |----------------------------|
    000A hex:|                              |
    0008 hex:|   Jump to DOS IOS calls      |
             |----------------------------|
    0007 hex:|   Jump to BASE               |
    0005 hex:|(the jump to DOS CPM calls)|
             |----------------------------|
    0004 hex:|   reserved for IOS           |
    0003 hex:|                              |
             |----------------------------|
    0002 hex:| Jump to IOS warm start       |
    0000 hex: ------------------------------
```

(Note that there is a data area within IOS that is reserved for the use of applications. This area is unique in that the memory contents remain intact across resets and warm starts. This can be useful for "chaining" programs. This area is at locations FF80 (HEX) through FFDF (HEX) inclusive.)

Applications programs interface to the IOS through three entry points only. These are locations 0000H, 0005H and 0008H. A discussion of each location now follows. (Note also that applications may also enter IOS routines through BOS calls. See section on BOS Calls elsewhere in this manual.)

## JUMP to LOCATION 0000 Hex

An application program that CALLs or JUMPs to location 0 will cause a warm start of the IOS. This CALL is used when an application program is finished running and wishes to return to whatever human interface program invoked it. In order to be compatable with CP/M, this entry point jumps to the WARMBOOT entry in a jump table which is identical to CP/M's BIOS Jump Table. It is recommended that applications programmers avoid attempting to use the BIOS Jump Table. The IOS is structured this way to be compatable with CP/M applications programs and to provide support expansion to the IOS.

## CALL to Location 0005 Hex

Location 0005 Hex is the same as the standard CP/M entry point. Details on this entry point are found in the section on CP/M Compatible Calls. Note that locations 6 and 7 contain a pointer to BASE, the first location used by the IOS. This allows applic-ations programs to determine how much memory is available. BASE may vary between different versions of the IOS.

## CALL to Location 0008 Hex

Location 0008 Hex is the entry point into the DOS IOS Calls. These calls are detailed in the section on DOS Calls. This entry point has the same calling conventions as the entry point at location 0005, except it is used for non-CP/M compatable operat-ing system calls. Note that locations 9 and A do NOT point to BASE.

When the MAIN MENU starts executing it will find the following initial conditions have been set:

- o The Stack Pointer is set to BASE
  (the first PUSH will write to BASE-1 and BASE-2)
- o All other Z-80 registers are undefined
- o All clock processing turned on
  - -Flashing Cursors Enabled
  - -Clock User Task Handling Enabled
  - -Real Time Clock Incrementing Enabled
- o The Video Chip is set to text mode.

- o Logical to Physical I/O Routing Set up to emulate Standard CP/M assignments:
  - -Video Device Location 1 set to:
    38 wide by 24 deep window with
    underline flashing cursor
  - -Console Output Routed to Video
    Device Location 1 (see window above)
  - -Console Input Routed to Human Interface
    Device Location 1 (Keyboard)
  - -List Routed to Printer
  - -Reader Routed to Human Interface
    Device Location 1 (Keyboard)
  - -Punch Routed to Video
    Device Location 1 (see window above)

After the MAIN MENU program gains control, it has the ability to alter the initial conditions for the application program which is to be loaded. For a complete list of the initial conditions as set up by the MAIN MENU program, please consult the Master Directory and Main Menu Specification 02-90020480.

### 4.1.1 Stack Operation and Requirements

The IOS only supports a single stack which is used by both the IOS and applications programs. This is different from CP/M which has two or more stacks, one or more used by CP/M and one for the application. Note that the IOS initializes the stack pointer to BASE so the stack will start at the highest available memory location and build down. The number of bytes of stack required by IOS depends on the number of peripheral devices attached. For the basic IOS, up to 64 bytes may be used by the operating software. This means an application program must be sure to allow for a stack size 64 bytes larger than what the application requires. The addition of peripherals to the NPC may increase the minimum stack requirement.

## 4.2 CP/M Compatible Calls

The IOS supports a number of calls which are similar to standard CP/M non-disk IO calls. No disk oriented calls are supported. The CP/M compatible calls that are provided are for resetting the system, and for performing IO from the logical devices CONSOLE, READER, PUNCH and LIST.

The IOS DOS CP/M compatable I/O facilities deal only with logical devices, (e.g. CONSOLE, LIST, READER, PUNCH). The IOS I/O Handlers operate with specific physical devices. (The IOS "attaches" the logical devices to the physical devices. For example, this allows an ASCII character to be sent to the logical device and it ends up at the physical device.)

The following logical devices are defined:

| | | |
|---|---|---|
| KEYBOARD: | (input portion of CONSOLE) | 0 |
| SCREEN: | (output portion of CONSOLE) | 1 |
| LIST: | (output) | 2 |
| READER: | (input device) | 3 |
| PUNCH: | (output device) | 4 |

The following physical devices are defined:

| | |
|---|---|
| HUMAN INTERFACE - | |
| KEYPAD: | 01 (input) |
| JOYSTICK 1: | 02 (input) |
| JOYSTICK 2: | 03 (input) |
| SCREEN WINDOW #1: | 11 (output) |
| PRINTER: | 21 (output) |

Assignments of physical devices to logical devices are performed by using the I/O Router Entry Point. When a program begins execution the following logical to physical attachments are made:

| LOGICAL | PHYSICAL |
|---|---|
| KEYBOARD | KEYBOARD |
| SCREEN | SCREEN WINDOW #1 |
| LIST | SCREEN WINDOW #1 |
| READER | SCREEN WINDOW #1 |
| PUNCH | SCREEN WINDOW #1 |

Note that SCREEN WINDOW #1 is defined by the system and is available to the application when it starts.

The CP/M compatible calls provided in the IOS through location 5 are as follows:

SYSTEM_RESET (call number 00H)
- performs same function as a jump to location 0000 Hex
- entry parameters:
  - C Register: 00 Hex
- is not re-entrant

CONSOLE_INPUT (call number 01H)
- reads the next character from the logical console with echo The call does not return until a character is ready. (This call will only accept CP/M compatible ASCII characters. If the "YES" key is hit, a "Y" is returned. If the "NO" key is hit a "N" is returned. All other key codes above 7FH are ignored.)
- entry parameters:
  - C Register: 01 Hex
- Returned Values:
  - A Register: Character Input
- is not re-entrant

CONSOLE_OUTPUT (call number 02H)
- outputs a character to the logical console (Since the default physical console driver is DOS call 0A2 and 0A3 consult the specification for DOS call A3H for control character interpretation.)
- entry parameters:
  - C Register: 02 Hex
  - E Register: Character to be output

READER_INPUT (call number 03H)
- get a byte from the logical TAPE reader control will not return to the calling program until the character has been read. (This call will only accept CP/M compatible ASCII characters. If the "YES" key is hit, a "Y" is returned. If the "NO" key is hit a "N" is returned. All other key codes above 7FH are ignored.)
- entry parameters:
  - C Register: 03 Hex
- returned value:
  - A Register: character read
- is not re-entrant

PUNCH_OUTPUT (call number 04H)
- output a byte to the logical TAPE punch (Since the default physical console driver is DOS call 0A2 and 0A3 consult the specification for DOS call A3H for control character interpretation.)
- entry parameters:
  - C Register: 04 Hex
  - E Register: character to be output

LIST_OUTPUT (call number 05H)

- output a character to the logical list device
- entry parameters:
  - C Register: 05 Hex
  - E Register: character to be output

DIRECT_CONSOLE_IO (call number 06H)

- provides unadorned I/O from/to the logical console

  Upon entry, the E register either contains an 0FF Hex, denoting a console input request, or a character to be output. If the input value if 0FF Hex, then the functions returns with the A register set to 00 if no character is ready at the logical otherwise the A register is set to the character value input from the logical console.

  (This call will only accept CP/M compatible ASCII characters. If the "YES" key is hit, a "Y" is returned. If the "NO" key is hit a "N" is returned. All other key codes above 7FH are ignored.)

  (Since the default physical console driver is DOS call 0A2 and 0A3 consult the specification for DOS call A3H for control character interpretation.)
- entry parameters:
  - C Register: 06 Hex
  - E Register: FF Hex (input) or character to be output
- returned value:
  - A Register: character of 00 Hex (input) nothing if output
- is not re-entrant

PRINT_STRING (call number 09H)

- print a string to the logical console from a buffer

  The character string stored in memory at the location pointed to by the DE register is sent to the logical console. A '$' is used as a delimiter to end the print string.

  (Since the default physical console driver is DOS call 0A2 and 0A3 consult the specification for DOS call A3H for control character interpretation.)
- entry parameters:
  - C Register: 09 Hex
  - DE Register: pointer to string

READ_CONSOLE_BUFFER (call number 0AH)
-read a line of editted logical console input to a buffer
The input is stored in the memory buffer pointer to by the DE register. If the buffer overflows console input is terminated. The format of the buffer is:

```
MAX_BUF_SIZE:                  BYTE;
NUMBER_OF_CHARACTERS_READ:     BYTE;
CHARACTER_BUFFER:              ARRAY[1..MAX_BUF_SIZE] BYTE;
```

The "GO" key (0D Hex) or CNTRL J (0A Hex) will terminate the input line. The DELETE key will delete the previosly typed character.
(This call will only accept CP/M compatible ASCII characters. If the "YES" key is hit, a "Y" is returned. If the "NO" key is hit a "N" is returned. All other key codes above 7FH are ignored.)
-entry parameters:
  C Register: 0A Hex
  DE Register: Pointer to MAX_BUF_SIZE
  (MAX_BUF_SIZE must be set as well)
-returned values:
  Console Characters in Buffer
  NUMBER_OF_CHARACTERS_READ set
-is not re-entrant

GET_CONSOLE_STATUS (call number 0BH)
-check to see if character has been typed at logical console
-entry parameters:
  C Register: 0B Hex
-returned value:
  A Register: 00 Hex -No character ready
  FF Hex -Character is ready and waiting
-is not re-entrant

I/O_ROUTER: ATTACH (call number 0AH)
-attaches a particular physical device to a logical device
-entry parameters:
  C Register: 0A Hex
  E Register: PHYSICAL_DEVICE
  D Register: LOGICAL_DEVICE

Where LOGICAL_DEVICE is the byte value of a logical device as identified in the section above and PHYSICAL_DEVICE is the byte value of a physical device. This call will cause all subsequent I/O to the logical device to be performed by the physical device attached. This call is available in the DOS.

## 4.2 INTRODUCTION TO DOS

The highest (and simplest) level of access to the IOS for applications programs is through the Downloadable Operating Software (DOS).

The entry points to the IOS use a standard calling convention and calling procedure. Each particular function is given a call number. This number is passed in the Z-80's C register. A function call may also accept zero, one, or two parameters as inputs and return zero or one value as an output. These parameters are passed as follows:

| | | | | |
|---|---|---|---|---|
| Function Number: | Passed in | C | register | if a BYTE |
| Return Value: | Returned in | A | register | if a BYTE |
| | Returned in | HL | registers | if a WORD |
| One Parameter: | Passed in | E | register | if a BYTE |
| | Passed in | DE | register | if a WORD |
| Two Parameters: | Passed in | E | register | if a BYTE |
| | Passed in | D | register | if a BYTE |

If more than 2 parameters need to be passed, then a dedicated data structure is implemented.

## 4.2.1 SEGMENT HANDLING ROUTINES

### 4.2.1.1 INTRODUCTION

The IOS provides the mechanism for interfacing with the data and programs that are found on the broadcast cycle. All data or code (program) which can be loaded at one time forms what is called a segment. Segment loads can be of varying size from a few bytes up to the NPC's available free RAM space. (By using segment load offsets, the application can manipulate data segments of much larger size.)

The interface that IOS provides is composed of two components. The first is that we provide DOS entry points which perform different segment handling functions. The second is that the IOS contains a data structure called the segment control/status block. This block of data is the place where data is passed to the segment handler and where data is received from the segment handler.

The following section will describe the theory or specification that the segment handler obeys. Following that are examples of how a programmer could use the segment handling functions.

### 4.2.1.2 SEGMENT CONTROL AND STATUS BLOCK

The IOS contains a data structure called segment control/status block. This block is used to pass information to and from the segment handler. This block resides inside the IOS and not in the application work space.

The programmer gains access to address of this block using DOS call 87H. By using a template of the control block as described below, the block can be modified as needed.

## CONTROL/STATUS BLOCK

| Field | Byte 1 | Byte 2 | Byte 3 |
|---|---|---|---|
| STATUS | | | |
| BYTES TRANSFERRED | LS | MS | |
| OPTIONS | | | |
| SEGMENT ADDRESS | MS | | LS |
| BUFFER POINTER | LS | MS | |
| BUFFER SIZE | LS | MS | |
| CONDITIONS | | | |
| OFFSET | LS | | MS |

Where:

STATUS:
- is a one byte variable
- is an output variable set by segment handler
- indicates the status of the segment operation as:

  1 busy doing operation
  0 operation finished with no error

MINUS NUMBERS - operation finished with error

- -1 tier not authorized
- -2 segment buffer overflowed
- -3 adaptor did not respond in time and segment handler timed-out
- -4 segment contained a bad packet
- -5 communication protocol failed between adaptor and P.C.

BYTES TRANSFERRED
- is a two byte variable least significant byte first
- is an output variable set by segment handler
- indicates number of bytes transferred into segment buffer

OPTIONS
- is a one byte variable
- is an input variable initialized by the application prior to segment operation
- indicates information on how segment is to be loaded:

If:

| | |
|---|---|
| bit 0=0 | control is returned immediately back to calling program after segment operation has started . |
| bit 0=1 | control is returned back to calling program after operation is finished. |
| bit 2=0 | data segments will be loaded into RAM. |
| bit 2=1 | data segments will be loaded into VRAM. |
| bits 3-7 | are reserved and should be 0. |

SEGMENT ADDRESS
- Is a 3 byte variable, most significant byte first.
- Is an input variable provided by the application, normally based on information from the directory.
- Indicates the segment identity to be loaded.
- This will be a number from 3 to 7FFFFFH.
- Is not required for all segment operations.

RAM POINTER
- Is a 2 byte variable, least significant byte first.
- Is an input variable provided by the application.
- Indicates where the segment or the status information is to be loaded. This would be some area inside the application or in VRAM.
- Is not required for loading segments where segment contains its load address.

RAM SIZE
- Is a 2 byte variable, least significant byte first.
- Is an input variable provided by the application.
- Indicates size of buffer in bytes; as pointed to by buffer pointer.
- Only required if buffer pointer is required.

CONDITIONS
- Returned. Can be ignored.

OFFSET
- This value (3 bytes) represents the number of bytes, from the beginning of the data segment, to ignore when loading the segment (an offset to the first loadable byte). Ensure that this is zeroed if you do not wish an offset.)

### 4.2.1.3 DOS INTERFACE

The segment handler performs operations based on the segment control block being correctly initialized, and a call being made to a DOS entry point.

*** NOTE *** In order for the segment loader IOS to properly interface with the Adaptor, the application program must NOT be within an interrupt protected area of code when making segment handler requests. Interrupts must be enabled and this implies that the call to the segment handler does not occur inside of a CRBEG - CREND code block. ( See "Interrupt Structure and Tasking Support" for more information on CRBEG, CREND and "Critical Regions". )

The DOS call is made by initializing register C and "calling" to location 8. The following calls will be the ones used by applications.

IF

| | |
|---|---|
| REG C = 80H | Segment handler is reset |
| REG C = 84H | Segment is loaded and interpreted if necessary |
| REG C = 87H | Base address of control block is read |

All parameters passed and returned are made through SEGCST.

SEGMENT HANDLER IS RESET     Call Number 80H

When this operation is invoked, any pending segment operation is ignored and the adaptor is reset to a known state. The segment control/status block does not have to be initialized because it is not used by this operation.

SEGMENT IS LOADED AND INTERPRETED IF NECESSARY.   Call Number 84H

This operation attempts to load in a segment as indicated by segment address in the control block. If the segment is loaded, the segment header may be interpreted to help with the load address and the location where execution of code is to begin or continue. If the load is unsuccessful, error information is returned in the status byte.

LOADING A DIRECTORY-ONLY SEGMENT

The control block requires that:

Options = 01 or 00
Segment address contains the number of a directory segment
Ram pointer and Ram size are not used

The segment will be loaded into the directory area inside IOS. The previous directory will be overwritten and the IOS will be notified that a new directory is present. The code-to-load field in the segment header will have been 000000 indicating that this directory has no code associated with it. After the directory has been loaded in, control is returned to the calling program.

LOADING A CODE SEGMENT:

The control block requires that:

Options = 01 {or 00}
Segment address contains the number code segment
Buffer pointer and buffer size are not used

The segment is loaded. The segment header contains the load address where the code is to be loaded. It also contains the start address where execution begins in the code after it has successfully loaded. Just prior to execution beginning in the newly loaded code, initialization occurs. The stack pointer is set to just below IOS, all attached tasks are removed, and the keyboard and clock interrupts are enabled.

LOADING A DIRECTORY WITH CODE-TO-LOAD

The control block requires that

Options = 01
Segment address contains the number of the directory segment
Ram pointer and Ram size are not used

The directory portion of the segment is loaded into the IOS directory area. Then the code-to-load field in the segment header is checked. If the value is FFFFFFH, then a code segment complete with header will immediately follow the directory in this same segment. If the value is not 0 and not FFFFFFH, then the code segment specified by the value is loaded in.

LOADING IN A DATA SEGMENT

The control block requires that:

Options = 01 or 00
Segment address contains the number of the data segment
Ram pointer contains the pointer to the area where the data is to be written
Ram size contains the size of the area where data is to be written
Offset is set to the number of bytes to ignore in the segment before loading (usually 0).

The data is loaded into the buffer as specified. After the data has been loaded control is passed to the calling program.

LOADING IN A OVERLAY SEGMENT:

The control block requires that:

Options = 1 or 00
Segment address contains the number of the overlay segment
Ram pointer and Ram size are not used

The overlay is loaded in at the load address specified by the segment header. After the segment has been loaded, control is returned to the calling program.

BASE ADDRESS OF SEGMENT CONTROL/STATUS BLOCK IS READ.
Call Number 87H

The application is returned to the base address of the control block in the HL register pair. This will allow the programmer to place a template of the control block at that address in order to initialize the block as required.

4.2.1.4 SEGMENT HEADERS

Each segment requires some overhead to describe what the segment contains. The extra data is called a segment header.

Segment headers have differing lengths. The minimum size of a header is 2 bytes long and the maximum size is 255 bytes.

The first byte of each header always contains the length of the header (2-255) in number of bytes.

The second byte of each header always contains the segment type. Four types of segments are currently defined:

Type 0 = Directory segment with or without code-to-load
Type 1 = Code segment
Type 2 = Data segment
Type 3 = Overlay segment

DIRECTORY SEGMENT HEADER

| Field | | | | | | |
|---|---|---|---|---|---|---|
| LENGTH | | | | | | |
| TYPE | 00 | | | | | |
| ENTRY WIDTH | | | | | | |
| NUMBER OF ENTRIES | | | | | | |
| CODE-TO-LOAD | MS | | LS | | | |
| NAME LENGTH | | | | | | |
| NAME | | | | | | |

Entry width is the width of the directory entries. Each directory entry has the same width. The width has a minimum size of 10 and maximum size of 255. (See section on directory calls.)

Number of Entries is the number of directory entries. This value could have a minimum of one and maximum of 255. However since the maximum directory is 1000 bytes, the product of entry width and number of entries must not exceed 1000.

Code-to-load is the field which indicates if code is to be loaded, and where this code can be found. If this value is FFFFFFH, then code follows the directory. All other values indicate the segment number of the code segment.

Name length is the number of characters in the name of this directory segment. The minimum is 1 byte and maximum is 255 bytes.

Name is the actual ASCII name of this directory segment.

## CODE SEGMENT HEADER

| Field | | | |
|---|---|---|---|
| LENGTH | 8 | | |
| TYPE | 1 | | |
| LOAD ADDRESS | MS | | LS |
| START ADDRESS | MS | | LS |

Load Address is a 3 byte variable, most significant byte first, which tells the segment handler where to place the code. Currently the first byte is always 0.

Start Address is a 3 byte variable, most significant byte first, which indicates where execution begins. Currently the first byte is always 0.

## DATA SEGMENT HEADER

| Field | |
|---|---|
| LENGTH | 2 |
| TYPE | 2 |

This header is the shortest one. Its purpose is to notify the segment loader that it is a data segment.

## OVERLAY SEGMENT HEADER

| Field | | | |
|---|---|---|---|
| LENGTH | 5 | | |
| TYPE | 3 | | |
| LOAD ADDRESS | MS | | LS |

Load Address is the address at which this overlay is to be loaded. It is a 3 byte variable, most significant byte first. Currently the first byte is always 0.

#### 4.2.1.5 EXAMPLES

In order to help illustrate the calls to the segment handler, examples written in assembler are included. They are arranged in some order from least difficult to more difficult.

>>>> ABORTING A SEGMENT LOAD <<<<

SCENERIO: The program has requested a large data segment, and decides that the data is not required. The segment load is aborted.

CODE:

```
LD      C, 80H
CALL    8
```

>>>> RESET SEGMENT LOAD DEVICE <<<<

SCENERIO: Although it is not required, a program may choose to reset the segment load device prior to loading a segment.

CODE:

```
LD      C, 80H
CALL    8
```

>>>> LOAD A DATA SEGMENT <<<<

SCENARIO: The program has determined that it needs to load data segment "tax-table". It has searched through its directory and found that "tax-table" is data segment no. 000234H. The table is to be loaded at location 8000H. Buffer is 1200H bytes long.

CODE:

```
LCB:                                  ; create a local
STATUS:    DB    0                    ; control block
BYTES:     DW    0                    ;
OPTIONS:   DB    1                    ; return control after load
                                      ; finished
SEG ADR:   DB    00,02H,34H           ; segment no.= 234
RAM PTR:   DW    8000H                ; buffer pointer
RAM SIZE:  DW    1200H                ; buffer size = 1200
CONDIT:    DB    0                    ; condition byte (returned)
OFFSET:    DB    0,0,0                ; load from beginning of segment
BASE:      DW    0                    ; value of base address
START:     LD    C,  87H              ; get base address of segment
                                      ; block
           CALL 8                     ;
           LD    (base), HL           ; temp. storage
           EX    DE, HL               ; DE=PTR to IOS control block
           LD    HL, LCB              ; HL=PTR to local control block
           LD    BC,BASE-LCB          ; # of bytes to move
           LDIR                       ; IOS control block initialized
                                      ;
           LD    C, 84H               ; load data file
           CALL 8                     ;
                                      ; check for error
           LD    HL, (base)           ; restore base address
           LD    A, (HL)              ; read status
           CP    0                    ; if status = 0
                                      ; then no errors occured
```

## >>>> LOAD AN OVERLAY SEGMENT <<<<

SCENERIO: The program requires that an overlay be loaded and a subroutine in the overlay executed. Overlay segment number is 54321 Hex.

CODE:

```
LCB:                                ; local control block
STATUS:     DB   0                  ;
BYTES:      DW   0                  ;
OPTIONS:    DB   1                  ; return control after load
                                    ; finished
SEGADR:     DB   5H, 43H,21H        ; seg number = 54321
RAM PTR:    DW   0                  ; don't care
RAM SIZE:   DW   0                  ; don't care
                                    ;
BASE:       DW   0                  ; storage for base address
                                    ;
START:      LD   C, 87H             ; get base address
            CALL 8                  ;
                                    ;
            LD   (BASE), HL         ; temp. storage
                                    ;
            EX   DE, HL             ; move local block
            LD   HL, LCB            ; to IOS block
            LD   BC, 7              ;
            LDIR                    ;
            LD   C, 84H             ; load overlay segm
            CALL 8                  ;
                                    ;
            LD   HL, (BASE)         ; check status
            LD   A, (HL)            ;
                                    ;
            CP   0                  ; if status NE 0
            JR   NZ, ERROR          ; then go to error
                                    ;
            CALL SUBROUTINE         ; else do subroutine
```

## >>>> LOAD A CODE SEGMENT <<<<

SCENARIO: A code segment is to be loaded in and executed. If an error occurs, during loading, execution is to re-boot the system. Segment number is 1234H.

CODE:

```
LCB:                               ; local control block
STATUS:    DB    0                 ;
BYTES:     DW    0                 ;
OPTIONS:   DB    0                 ; return control
                                   ; immediately
SEG ADR:   DB    0, 12H, 34H       ; seg no. 1234
RAM PTR:   DW    0                 ; don't care
RAM SIZE:  DW    0                 ; don't care
                                   ;
START:     LD    C, 87H            ; get base address
           CALL  8                 ;
                                   ;
           EX    DE, HL            ; move local block
           LD    HL, LCB           ; to IOS
           LD    BC, 7             ;
           LDIR                    ;
                                   ;
           LD    C, 84H            ; load code segment
           CALL  8                 ;
                                   ;
           JP    0                 ; error  occurred, reboot
```

## >>>> LOADING A DIRECTORY SEGMENT WITH NO CODE TO LOAD. <<<<

Example is identical to loading an overlay segment.

## >>>> LOADING A DIRECTORY SEGMENT WITH CODE TO LOAD. <<<<

Example is identical to loading a code segment.

## 4.2.2 DIRECTORY ROUTINES

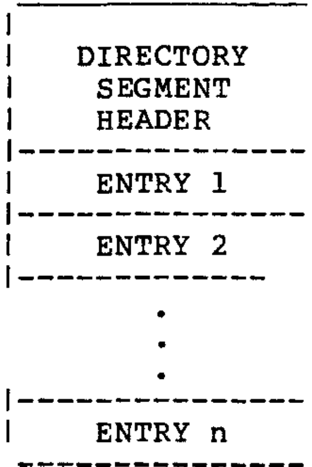
### 4.2.2.1 INTRODUCTION

Each application which requires segments off the cable will have a segment directory loaded into memory. The segment directory will be stored as a 1K buffer as part of the IOS. Each entry of the directory will contain information about one segment being transmitted on the cable.

This section describes the format of the directory in memory, and the method that applications use to access information in the directory.

### 4.2.2.2 FORMAT OF DIRECTORY

The segment directory is stored in a 1K buffer in the IOS. The format of the information in the buffer is as follows:




The DIRECTORY SEGMENT HEADER is the standard segment header as described in section 4.2.1.2.3. Following the header, are a number of entries, one entry per segment in the directory.

Each directory entry has the following format:

| Field | Size |
| --- | --- |
| TYPE | 1 BYTE |
| OWNER | 1 BYTE |
| TIER LEVEL | 4 BYTES |
| SEGMENT ADDRESS | 3 BYTES |
| NAME | 18 BYTES |
| RESERVED | 4 BYTES |

TYPE and OWNER are each one byte values that are currently undefined in the NABU NETWORK. TIER LEVEL is a four byte value which gives the tier access information of the segment, each bit corresponding to a different tier level. The SEGMENT ADDRESS is a three byte value which gives the address number of the segment on the cable. This three byte value is the number that must be put into the SEGMENT ADDRESS of the SEGMENT CONTROL AND STATUS BLOCK when a request is made to load the segment off the cable (see section 4.2.1).

The NAME of the segment is given by the applications programmer when submitting the segment to the APS. It can be up to 18 characters long. In the directory entry the name is left justified and right padded with blanks. In most cases the application will know the name of the segment to be loaded, and search through the directory to find the segment address in order to request the segment off the cable.

The last four bytes of each directory entry are reserved for system use and should not be used by the application.

### 4.2.2.3 ACCESSING THE DIRECTORY

Multi-Segment applications require a means of loading overlays or data. The segment loader requires that a segment number or address be present in the segment control block. The user directory contains the information which links the segment name with the segment address. This directory is loaded into an IOS directory area. This is an internal 1K buffer.

An application has one DOS calls available for accessing the directory: the routine to search through the directory for a particular entry.

DIRECTORY SEARCH — DOS CALL 88H

PURPOSE: To search for a particular entry in the directory.

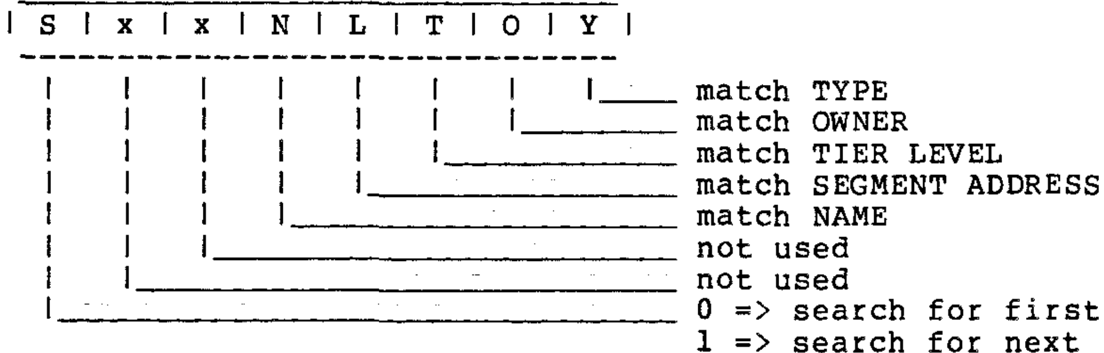
PARAMETERS PASSED: C Register - 88H
DE Register - Address of a Directory Search Block (see below).

PARAMETERS RETURNED: All information returned is done so in the Directory Search Block as described below.

DATA STRUCTURES: The Directory Search Block is a data structure declared by the application and passed to the IOS when the directory search call is made. the Directory search Block has the following format:

| Field | Size |
|---|---|
| MATCH PATTERN | 1 BYTE |
| ENTRY WIDTH | 1 BYTE |
| TYPE | 1 BYTE |
| OWNER | 1 BYTE |
| TIER LEVEL | 4 BYTES |
| SEGMENT ADDRESS | 3 BYTES |
| NAME | (ENTRY WIDTH) - 9 BYTES |

The MATCH PATTERN is set by the calling application and is used to indicate which fields in the directory entry are to be searched for. The meaning of each of the bits in the MATCH PATTERN byte is as follows:



If the N, L, T, O or Y bits are set in the SEARCH BLOCK, this indicates that the corresponding fields of the search block are to be matched with the directory entry. For example, if the application wished to search for a segment with the name COSMOS, it would set the N bit, and reset the L, T, O, and Y bits. If the application wished to search for a segment with the name COSMOS <u>and</u> with owner 33, it would set the N and O bits, and reset the L, T, and Y bits.

The S bit of the match pattern indicates whether to search for the first directory entry which matches, or the next directory entry after the last search.

The ENTRY WIDTH gives the number of bytes in the search block following the ENTRY WIDTH byte. It is required to compute the number of characters in the NAME.

The fields TYPE, OWNER, TIER LEVEL, and NAME correspond to the fields in the directory entry. The application will set these fields if it wishes to search the directory for a corresponding entry. For example, if the application wished to search for a segment with owner 33, it would set OWNER to 33; if it wished to search for a segment called NEUTRON BOMB it would set NAME to 'NEUTRON BOMB'.

When matching on NAME the application can use "wildcard" features. A '?' with the high bit set (i.e., 0BFH) will match any single character in the corresponding position. For example if NAME is two bytes long and is set to 41H, 0BFH, the directory search routine will match A1, A2, AN, or any other segment whose name starts with A. A '*' with the high bit set (i.e., 0AAH) will match any string in the corresponding position. For example if NAME is two bytes long and is set to 41H, 0AAH, the directory search routine will match on A, AA, A1, A12, A123, AAASSSDDD, or any segment name beginning with 'A'.

The matching of the tier level fields is done in the same manner as the tier authorization match in the Adaptor. That is the two fields are ANDed together. If the result is non-zero then there is a match.

All values returned to the calling application are done so in the SEARCH BLOCK in the following manner:

- If the search was successful the MATCH PATTERN is set to 0FFH by the search routine.
- If the search failed then the MATCH PATTERN is set to 0 by the search routine.
- If the search was successful, then the TYPE, OWNER, TIER LEVEL, SEGMENT ADDRESS, and NAME fields are filled in to correspond to the entry in the directory that was found.

The following are a few examples of how the directory search routines work.

```
;Read in a segment named BOMBAST
;
;Search the directory for the first entry with the name BOMBAST
;
     LD   A,10H
     LD   (MATCH_PATTERN),A    ;Search for first occurrence.
                               ;Match on name.
     LD   A,23
     LD   (ENTRY_WIDTH),A      ;Set entry width

     LD   HL,SEGNAME
     LD   DE,NAME
     LD   BC,14
     LDIR                      ;Copy BOMBAST into NAME field.

     LD   C,88H
     LD   DE,SEARCH_BLOCK
     CALL 8                    ;Call IOS to search the directory.

     LD   A,(MATCH_PATTERN)
     OR   A
     JP   Z,NOT_FOUND          ;Was search successful?

;Search was successful.  Read in segment.
;Segment address is in SEG_ADDR
     CALL READSEG
          .
          .
          .
          .
```

```
;Read in second segment in directory which begins with the
;letter 'A' and is owned by 69.  This will require doing two
;directory searches; one for the first occurence; and one
;for the next occurence.
     LD   A,12H
     LD   (MATCH_PATTERN),A      ;Find first occurence.  Match on
                                 ;NAME and OWNER.

     LD   A,23
     LD   (ENTRY_WIDTH),A        ;Set ENTRY_WIDTH to 23

     LD   A,'A'
     LD   (NAME),A               ;First character of NAME is 'A'.

     LD   A,0AAH
     LD   (NAME + 1),A           ;Wildcard feature

     LD   B,12
     LD   A,' '
     LD   HL,NAME + 2

BLANK: LD (HL),A
     INC  HL
     DJNZ BLANK                  ;Blank out remainder of NAME

     LD   C,88H
     LD   DE,SEARCH_BLOCK
     CALL 8                      ;Call directory search

     ;Search for first is done.  If successful then search for next.
     LD   A,(MATCH_PATTERN)
     CP   0
     JR   Z,NOT_FOUND            ;Was search successful

     LD   A,92H
     LD    (MATCH_PATTERN),A      ;Search for next occurence of
                                  ;NAME and OWNER

     LD   A,0AAH
     LD   (NAME + 1),A           ;Wildcard feature

     LD   B,12
     LD   A,' '
     LD   HL,NAME + 2
```

```
BLANK1: LD (HL),A
        INC  HL
        DJNZ BLANK1                   ;Blank out remainder of NAME

        LD   C,88H
        LD   DE,SEARCH_BLOCK
        CALL 8                        ;Call directory search

        LD   A,(MATCH_PATTERN)
        CP   0
        JR   Z,NOT_FOUND
        ;Read in the segment
        CALL READSEG
             .
             .
             .
   SEARCH_BLOCK:
   MATCH_PATTERN:       DS    1
   ENTRY_WIDTH:         DS    1
   TYPE:                DS    1
   OWNER:               DS    1
   TIER_LEVEL:          DS    4
   SEG_ADDR:            DS    3
   NAME:                DS    14
```

### 4.2.3 The Interrupt Structure and Tasking Support

#### 4.2.3.1 Introduction

Because of the Real-Time requirements of the NABU P.C., some sort of real time operating support is required. Because of the overhead involved in context switching and maintaining task descriptors and the questionable utility to applications programs, full multi-tasking is not supported. Instead, a modified forground/background tasking approach is used. The application program runs as the foreground task, and is in complete control of the NPC. The application program may also call IOS routines which run in the foreground, and may also start other tasks running in the background. The application program may also use IOS routines to start applications tasks running in the background.

The background tasks are always started by the occurance of interrupts and must "run to completion". The NPC supports eight vectored, maskable, nestable, priority interrupts. Each interrupt has an interrupt service routine or "system task" associated with it. Some of the interrupts may also have one or more "user tasks" associated with it. These "user tasks" will start after the system task for the given interrupt has completed. The NPC interrupts in order of priority are:

1. NNI Receive
   -activated when a character is received from the NNI
   -system task for packet/segment reception handling attached to this interrupt
   -No user tasks may be attached

2. NNI Send
   -activated when a character has been sent from the NPC to the NNI (Transmitter Buffer Empty)
   -System task for packet/segment reception handling attached to this interrupt
   -No user tasks may be attached

3. Human Interface Input
   -activated when a character has been received from the remote keyboard

4. Video Frame Sync (60 Hz Clock)
   - -activated every 1/60 sec by start of vertical retrace on the TMS 9918A Video Display
   - -System task to: flash cursors, update real time clock, etc. etc attached to this interrupt
   - -System task to timeout on NNI response attached to this interrupt
   - -Any number of user tasks may be attached.

5. Option Card Interrupt from Slot No. 0
6. Option Card Interrupt from Slot No. 1
7. Option Card Interrupt from Slot No. 2
8. Option Card Interrupt from Slot No. 3

Option Card Interrupts
- -activated by option cards
- -one system task per card may be attached as required
- -one user task per card may be attached

### 4.2.3.2 Critical Regions

For the purposes of the IOS, a critical region is defined as a section of executable code, or data structure which may be accessed by only one concurrently executing task at a time. Critical regions are bound to exist in any system which supports more than one concurrently executing task.

Two IOS BOS calls are used to protect critical regions in the IOS and in applications programs. When entring a critical region, an application task must call the routine "CRITICAL_REGION_BEGIN". This is call number 02 in the IOS BOS and its assembly language name is CRBEG. It takes no parameters. When leaving the critical region the routine "CRITICAL_REGION_END" must be called. This is call number 03 in the IOS BOS and its assembly language name is CREND. No registers are destroyed by these calls.

Critical regions are nested by CRBEG and CREND. This nesting is analogous to opening a left bracket for each CRBEG that is performed and closing the critical region with a right bracket each time a CREND is performed. In this way it is easy to visualize that one may have a critical region within a critical region and that interrupts will only be enabled when the final right bracket (CREND) is reached. It is also obvious that there must be as many CRENDs as there are CRBEGs in order to keep the interrupt control in order.

Since all interrupts are disabled inside critical regions, they MUST be kept as SHORT as possible.

*** NOTE ***

Attempting to interface with the segment handler while in a critical region, may yield unpredictable results. Avoid this situation.

It is also strongly recommended that applications not use the EI and DI assembler instructions for critical region protection. Use the IOS CRBEG / CREND routines instead. (See BOS Calls.)

### 4.2.3.3 User Task Attachment Routines

#### 4.2.3.3.1 Attaching Tasks to the Clock

An application program may attach tasks to the TMS-9918A VDP frame interrupt. As many tasks as desired may be attached. Tasks may be attached and removed from the clock by both the main application program and by other tasks. In fact a task may remove itself from the clock. It is also possible to have multiple invocations of the same piece of code as separate tasks.

BEFORE CLOCK-ATTACHED TASKS ARE EXECUTED, ALL REGISTERS ARE SAVED, AND THEY ARE RESTORED AFTER THE COMPLETION OF THE TASK.

A data structure called a TASK_CONTROL_BLOCK (TCB) is used to keep track of the relevent parameters of a task which is attached to the clock. The TCB has the following structure:

```
VAR
  TASK_CONTROL_BLOCK: RECORD OF
                      BEGIN
                        NEXT_BLOCK          : WORD
                        RESET_INTERVAL      : BYTE
                        CURRENT_INTERVAL    : BYTE
                        TASK_ADDRESS        : WORD
                      [ PARAMETER_BLOCK     : USER_DEFINITION ]
```

The NEXT_BLOCK word is used by the operating system to place the TCB on a linked list with other TCB's. This word should not be altered by applications tasks at any time.

The CURRENT_INTERVAL byte counts the number of ticks that have gone by since the last time the task was activated. It is accessed by the IOS but may also be accessed by applications tasks. The IOS algorithm in which this byte is used is as follows:

```
ON EACH CLOCK INTERRUPT DO
   BEGIN
      FOR EACH TASK_CONTROL_BLOCK DO
             BEGIN
              CURRENT_INTERVAL = CURRENT_INTERVAL - 1;
              IF CURRENT_INTERVAL = 0 THEN DO
                BEGIN
                   CURRENT_INTERVAL := RESET_INTERVAL;
                   RUN_THIS_TASK( TASK_ADDRESS);
                END
              END
   END.
```

The byte CURRENT_INTERVAL is decremented every clock tick until it equals zero. When CURRENT_INTERVAL = 0, the task is executed. Therefore an INTITIAL DELAY may be issued before the task is dispatched by initializing CURRENT_INTERVAL to a value greater than one. Before execution of the task, CURRENT_INTERVAL is reset to the value of RESET_INTERVAL. CURRENT_INTERVAL is measured in clock ticks which are approx 1/60 of a second long. For example a value of 5 means the task will every 5/60 of a second and a value of 1 means the task will run every 1/60 of a second or 16 milliseconds.

> NOTE: That initializing CURRENT_INTERVAL to zero will cause the task to be delayed for 256 clock ticks (approx 4 seconds) before it is executed a first time.

The CURRENT_INTERVAL byte can be used to determine when a task has last run or when a task will next run. It will also determine when a newly created task will next run.

The RESET_INTERVAL byte is the value to which the byte CURRENT_INTERVAL is initialized to after RESET_INTERVAL has been decremented to zero. This byte is never changed by the operating system, but can be changed for purposes of changing the re-execution time of an active task.

The TASK_ADDRESS word contains a pointer to the start of the task or user subroutine. When control is given to the interrupt subroutine, the pointer to the TASK_CONTROL_BLOCK (TCB) is in the BC register so that the user may access any of the bytes in the TCB and modify them if he so desires. NOTE also that this pointer is useful for accessing variables (bytes or words) immediatly below the 6 byte TASK_CONTROL_BLOCK.

Also note that an interrupt subroutine or task should always end with a RETURN (RET) statement in order to return control to the application's mainline and never jump out of a user task otherwise interrupts will remain permantly disabled. Interrupt subroutines should also always be as short as possible and never take longer than 16 milliseconds (1 CLOCK TICK) to execute. If it should be neccessary to run a task that takes longer than 16 miliseconds, it should be broken up into two tasks which execute on alternate interrupts. To do this start one task immediatly and delay the second task one clock tick by setting CURRENT_INTERVAL initially to two.

The PARAMETER_BLOCK is an optional data structure which may be accessed by an applications task. When a task is started the address of the TASK_CONTROL_BLOCK (ie a pointer to the NEXT_BLOCK word) is passed in the BC Register. This gives the task access to its own TCB. Using the parameter block to keep all of the task's data will allow several instantiations of the same code as separate tasks without resorting to keeping data on stack frames.

There are a few important calls which should be mentioned at this point, because tasks may not run if they are not done.

A call must be done at the start of every program to link in the BOS routines so that calls to CRBEG and CREND will work. (See section on BOS Calls for information on linking BOS routines using DOS call 90H.)

```
ex:
     MAIN::
          LD C,90H          ;DOS CALL 90H
          LD DE,LNKTB##     ;ADDRESS  OF
                            ;LINK TABLE
          CALL DOS          ;CALL SYSTEM
```

A call must be done to CLKPRM (BOS CALL #37) should be done to enable user task dispatching when the user is ready to have the tasks dispatched.

```
ex:       LD C,4            ;CONSIDER CLOCK
                            ;TASK
                            ;DISPATHING BIT
          LD E,4            ;SET BIT #2  TO
                            ;TURN ENABLE
                            ;TASK DISPATCHING
          CALL  CLKPRM##    ;NOW TASKS WILL
                            ;RUN IF INTERRUPTS
                            ;ARE ENABLED
```

NB: a call to CLKPRM should be used instead of a call to CRBEG if the user wishes to disable ALL tasks for a long period of time. Because a call to CRBEG will disable ALL interrupts, and not only user tasks.

The IOS DOS Routines which support attaching and removing clock tasks are as follows:

CLOCK_USER_TASK_ATTACH (DOS call number 8BH)
- used to attach a user task to the clock ISR
- entry parameters:
  - C Register: 8B Hex
  - DE Register: Pointer to a Task Control Block

CLOCK_USER_TASK_REMOVE (DOS call number 8CH)
- used to remove a user task from the clock ISR
- entry parameters:
  - C Register: 8C Hex
  - DE Register: Pointer to a Task Control Block
    If DE = 0, then all tasks are removed

Note that although there is no limit to the number of tasks which may be attached to the clock, attaching too many tasks, or attaching long running tasks may cause clock interrupts to be lost.

EXAMPLE

This routine demonstrates attaching and removing tasks.

```
        DOS    EQU  0008      ; DOS ENTRY POINT
        COUNT: DB   0         ;Data byte incremented every 5/60
                              ;a second by interrupt subroutine
                              ;COUNTER
        CNTTCB:               ;TASK_CONTROL_BLOCK
               DW   0         ;point to next  block  = NIL
               DB   5         ;RESET_INTERVAL
               DB   100       ;INITIAL_DELAY Of 100 ticks
               DW   COUNTER   ;Interrupt subroutine

 MAINLINE::              LD  C,90H        ;LINK_BOS_ROUTINE
                                          ;call number
                         LD  DE,LNKTB##   ;address of user
                                          ;LINKTABLE
                         CALL DOS         ;call location 8
                         .
                         .                ;SOME MORE MAINLINE CODE
                         .
                         LD C,08BH        ;C REG = 08BH = CLOCK_USER_ATTACH.
                         LD DE,CNTTCB     ;DE REG = ADDRESS of TCB
                         CALL  DOS        ;CALL location 8 to attach task
                                          ;Now task is ATTACHED!!
                         .
                         .                ;MORE MAINLINE SETUP
                         .

        DRIVER::                          ;MAINLINE DRIVER
                  LD A,(COUNT)            ;get count in A reg.
                  CP 100                  ;Is count = 100?
                  JP NZ,DRIVER            ;NO. Then wait till
                                          ;COUNT = 100
                  LD A,0                  ;YES. count = 100 so
                  LD (COUNT),A            ;then reset count to 0
                  LD C,8CH                ;USER_TASK_REMOVE
                  LD DE,CNTTCB            ;TCB ADDRESS
                  CALL DOS                ;Now counting task will
                                          ;no longer and the byte
                                          ;COUNT will no longer be
                                          ;incremented.
                  JP DRIVER               ;and do forever.

            COUNTER:                      ;USER'S INTERRUPT SUBROUTINE
                  LD A,(COUNT)            ;get current count
                  INC A                   ;increment count
                  LD (COUNT),A            ;store new count
                  RET                     ;return to mainline.
```

### 4.2.3.3.2 Keyboard User Tasks

The IOS DOS call 95H will permit tasks to be attached to the keyboard. The user passes, in the DE registers, a pointer to a table whose format is as follows:

```
 ----------------
|   Number of    |
| Entries (Byte) |        /     ---------------------------
 ----------------         |    |   Character Code (Byte)   |
|   ENTRY #1     |        |     ---------------------------
 ----------------         |    | SYM Key Qualifier (Byte)  |
|   ENTRY #2     | -------      ---------------------------
 ----------------         |    |        Pointer to         |
        .                 |    |    User Routine (Word)    |
        .                 \     ---------------------------
        .
 ----------------
|   ENTRY #n     |
 ----------------
```

The character code is the code sent by the keyboard. Joystick data is ignored. The SYM key qualifier indicates when the task is to be performed as follows:

```
bit 0 =   0  then do not execute user routine when  SYM
             key is down.
    0 =   1  then execute user routine when SYM key is
             down.

bit 1 =   0  then do not execute user routine when SYM
             key is up.
    1 =   1  then execute user routine when SYM key is
             up.
```

The user's routine must end with a RET to prevent disaster.

If a user's routine is to be executed, then all normal processing of the key code received is superseded. This means the key will not be put in the queue and the PAUSE and TV/NABU keys will not be processed - only the user routine will be performed. A number of tasks may be attached to the same keyboard input code. This allows the application the option of having different tasks execute based on the condition of the SYM key qualifier for the same keyboard input code.

Attaching a task to the SYM key may produce unusual results. This is due to the fact that the attached task will not allow the SYM key to execute in the proper manner.

WARNING - When attaching tasks to the keyboard that will attempt to write to the video control register ensure that no other foreground or background task is using the video routines FASTLD and FASTDU (and their 256 byte cousins) as they allow keyboard interrupts to occur.

To remove the keyboard task table in its entirety from the keyboard, DOS call 95H is performed with the DE registers set to zero.

### 4.2.3.3.3 EXPANSION SLOTS

Expansion slots in the NABU PC will be used to allow a number of different option cards to be added. There are four expansion slots in each PC. The option cards send an identification (id) code to a port at the option slot. The NABU PC picks up the ids by specifying the hex value C0H for slot 0, D0H for slot 1, E0H for slot 2, and F0H for slot 3. Since there can be so many different cards which can be installed in the NABU PC, and different configurations of these cards in the slots, it is not reasonable to include drivers for each option card in IOS.

The solution is to have the application identify the cards installed in the expansion slots, and have interrupt service routines which will handle the option cards.

To find out what is in the expansion card slots, a DOS call 94H, (GET_CONFIG) can be made. The input parameter is 94H passed in the C register. This call returns the address of the configuration block in registers HL. The format of the block is as follows;

```
CONFIGURATION BLOCK
     STRUCTURE ( IOS_VERSION_NO    BYTE,
                 IOS_LEVEL_NO      BYTE,
                 RESERVED          WORD,
                 SLOT_0_CONTENTS   BYTE,
                 SLOT_1_CONTENTS   BYTE,
                 SLOT_2_CONTENTS   BYTE,
                 SLOT_3_CONTENTS   BYTE )
     END STRUCTURE
```

An interrupt service routine can then be attached to an option slot interrupt letting the application deal with the option card directly.

DOS call 8DH is the Slot Interrupt Service Routine Attach. The entry parameters are:

```
     reg  C: 8DH
     reg DE: pointer to ISR Control Block
               where the ISR Control Block contains
                    byte 1     - slot number (C0,D0,E0, or F0 cor-
                                 responding to slots 0,1,2 or 3)
                    byte  2,3  - pointer  to start  of  interrupt
                                 service routine.
```

The address of the ISR is placed into the interrupt vector table by the IOS.

Dos call 8EH is the Slot Interrupt Service Routine Remove. The parameters that must be passed to this routine are:

```
Reg  C = 8EH
Reg  E = Slot number (C0,D0,E0,F0  corresponding  to
         slots 0 to 3).
```

This routine disables interrupts from the slot and then removes the address of the ISR from the interrupt vector table.

The applications programmer must know the identification codes which are sent by the different option cards which the application will be using. The programmer must also initialize the interrupt hardware on the option card (if applicable).

## 4.2.4 HUMAN INPUT

### 4.2.4.1 INTRODUCTION

This section explains how keyboard and joystick data may be accessed through the IOS.

### 4.2.5.2 SPECIAL KEY OPERATION

Several keys have special reserved functions and the IOS traps and handles these keys:

EXIT OPERATION:
PAUSE OPERATION:
TV/NABU SWITCH:
SYM OPERATION:

The Exit Operation simply consists of jumping to location 0000H. This will cause a system re-boot to occur. (See also the section on XIOS.)

The Pause operation stops the execution of the applications program. A LED on the NPC front pannel is turned on to indicate that the NPC is in Pause mode. While paused, only the SYM, EXIT, and PAUSE operations are interpreted. All other keys and human interface inputs are ignored. Pause mode is quit either by the reset operation or by another pause operation.

The TV/NABU switch is used to switch between the external video input and the NPC generated video. When the NPC is booted NPC video is switched in. When the NPC is powered off, the hardware ensures that the external video is switched in. At any time when the NPC is operating the TV/NABU switch may be used to switch between computer generated and external video sources.

For details as to what keycodes constitute the EXIT, PAUSE, TV/NABU and SYM operations, see TABLE 1.

The IOS handles all SYM key operation. See section 4.2.5.5 for more information.

PUT CHARACTER CODE TABLE HERE

### 4.2.4.3 OBTAINING DATA FROM THE KEYBOARD

The keyboard device driver has two entry points which are set up as standard serial device drivers. They are as follows:

The routine "HUMAN_INPUT: DEVICE_READY" can be called to see if a particular keyboard device has data available. This call returns 0 if nothing is ready and some non-zero value if there is a character ready. Note that the keyboard unit only sends joystick data if the value changes from the previous reading. Also, the keyboard unit "de-bounces" digital joystick data. This means that if HUMAN_INPUT: DEVICE_READY returns TRUE for a particular joystick port, the value for that device is guaranteed to have changed.

The parameter passing for the human interface is as follows:

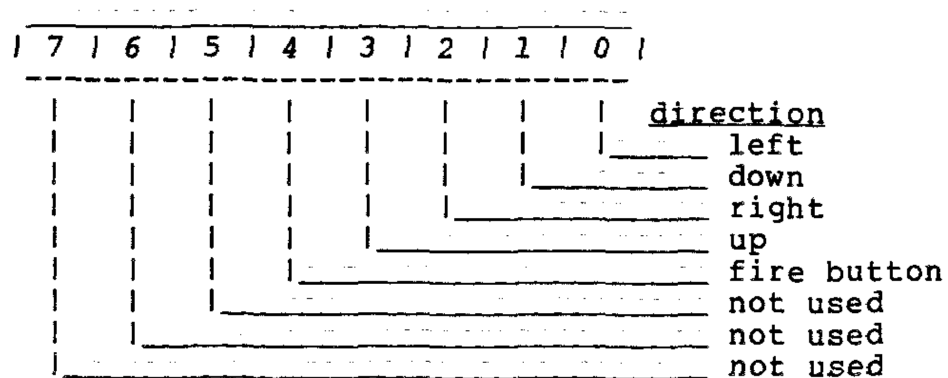
```
HUMAN_INPUT: DEVICE_READY                 (call number A0H)
     -returns a data ready indication for a specified human
      interface input
     -entry parameters:
        C  Register: A0 Hex
        E  Register: device location to be checked
     -returned value:
        A  Register: 00 if device not ready
                     non-zero value if device is ready

HUMAN_INPUT: GET_DATA                     (call number A1H)
     -gets a data byte from a specified human interface input
     -see section 3.3.7.1
     -entry parameters:
        C  Register: A1 Hex
        E  Register: device location to get data from
     -returned value:
        A  Register: data input from human input device
```

Both routines pass the device location in the E register. The following device locations are defined:

| | |
|---|---|
| 00H | -Reinitializes the keyboard device driver by making all devices "not ready" (throws away any ready data) (Works with DEVICE_READY only) |
| 01H | -Keyboard |
| 02H | -Joystick 1 |
| 03H | -Joystick 2 |
| FFH | -returns the base address of the current SYM key re-definition table. (returns address in HL) (Works with DEVICE_READY only) |

The values returned from DOS call A1 are either joystick data, or keyboard data.

Joystick data uses the first five bits of the byte to determine the joystick's new change of direction.




Keyboard data is sent as single 8 bit bytes, usually in an ASCII format. There are however function keys which transmit special byte values. Table 1 should be consulted.

### 4.2.4.4 Set SYM Table

The IOS performs all SYM key decoding. A 128 character look-up table, is maintained if it has been defined by an application program and passed to the IOS. Any key which is NOT release coded may have its meaning changed by holding down the SYM key while the key is pressed. A new key-code is chosen by doing a look-up in the re-definition lookup table. The resulting value is then passed on to the application program. If the SYM key is pressed when there is no defined redefinition table, then the ASCII value of the key pressed with the high bit set is passed on to the application program.

SYM key re-definition is NOT performed on any key which is release coded. Release codes are sent onto the application by the device handler. It is up to the application to ignore them if they are not desired.

The call SET_SYM_TABLE (call number 91H) is used to set the SYM *redefinition table base address.* The base address of the redefinition table is passed as a parameter. If the address passed is 0000H then any redefinition table currently in use is freed, and the new redefinition consists of setting the high bit in the ASCII code. The format of this call is as follows:

```
SET_SYM_TABLE                              (call number 91H)
     -used to set the SYM key Redefinition table
     -entry parameters:
        C  Register: 91 Hex
        DE Register: Pointer to new SYM key table
                           where the SYM KEY TABLE has the
                           following format:
                                128 Entries each one byte long
                                    Entry 0 contains the
                                        redefinition code of
                                        keyboard input code 0 when
                                        SYM key is down. The
                                        redefinition code is placed
                                        in the keyboard buffer or
                                        queue.
                                    Entry 1 contains the
                                        redefinition code of
                                        keyboard input code 1 when
                                        SYM key is down.

                                              .
                                              .
                                              .
```

Entry 7FH contains the redefinition code of keyboard input code 1 when SYM key is down.}

-returned value:
HL Register: Old SYM key table

The SYM Key Redefinition table is 128 bytes long. The contents of this table are used to redefine or translate the received ASCII character (values 0 to 7FH) into a different ASCII character. For example, if the first entry in the table is 7FH (delete character), and an ASCII 0 (ctrl @) is received, the CTRL @ will be replaced with the delete character. See TABLE 1 for the Keyboard ASCII Code Chart.

### 4.2.5 Video Screen Device Driver

In keeping with the standard for physical device drivers, two entry points are provided for the Video Screen Device Drivers. These are as follows:

VIDEO_SCREEN: DEVICE_READY (call number A2H)
- returns a data ready indication for the video screen driver.
- entry parameters:
  - C Register: A2 Hex
- returned value:
  - A Register: 00 if device not ready<br>non-zero value if device is ready

VIDEO_SCREEN: SEND_DATA (call number A3H)
- writes a character to the specified window
- entry parameters
  - C Register: A3 Hex
  - D Register: data to be output
- returned value:
  - A Register: 00 if device not ready<br>non-zero value if data was sent

- It will handle control characters: carriage return, line feed, delete, backspace, form feed, and horizontal tabs. The routine puts the character at the current cursor position. Bit 7 is stripped off each ASCII character by "anding" with 7FH prior to displaying. It will interpret the control characters as follows:

LINE FEED: CONTROL J
If the cursor is on the bottom line of the window, the window will scroll up one line and leave the bottom line filled with SPACES and the cursor will drop straight down into this blank line. If the cursor is in the middle of the window, the cursor just drops down one line.

CARRIAGE RETURN: CONTROL M
The cursor will move to the first position of the current line.

BACKSPACE: CONTROL H
The cursor moves back one position. If the cursor is in the top-left position of the window, nothing happens.

DELETE: 7FH
The cursor backspaces one character and places a SPACE over the character.

FORM FEED: CONTROL L
The cursor is reset to the top-left position of the window and the window is filled with SPACES.

BELL: CONTROL G
A short tone will sound.

VERTICAL TAB: CONTROL K
The cursor moves up one line. If the cursor is on the top-most line, nothing will happen.

HOME: CONTROL ^
The cursor is reset to the top-left position of the window.

OTHER CONTROL CHARACTERS:
Nothing will happen.

### 4.2.6 Printer Output

The printer output devices allow for data to be sent to a printer connected to the personal computer. There are two calls available.

DOS call 0A4H determines whether the printer is ready to receive data. A non zero value will be returned if the printer is not ready.

DOS call 0A5H will perform wait until the printer is ready and then send the data to the printer. The appropriate register values for the DOS calls A4 and A5 are:

```
PRINTER_OUTPUT: DEVICE_READY            (call number A4H)
     -returns a printer ready indication
     -entry parameters:
        C  Register: A4 Hex
        E  Register: device location to be checked
     -returned value:
        A  Register: 00 if device not ready
                     non-zero value if device is ready


PRINTER_OUTPUT: SEND_DATA               (call number A5H)
     -writes a character to the printer
     -entry parameters:
        C  Register: A5 Hex
        E  Register: Device location where data is to be
                     sent
        D  Register: Data to be output
```

EXAMPLE

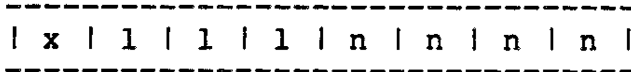
The following Z80 assembler example demonstrates how to print a form feed on a printer.

```
FF     EQU     0C

START:   LD C,0A4H        ;PRINTER OUTPUT
         LD D,FF          ;LOAD THE DATA TO BE DISPLAYED
         LD E,02H         ;THE PRINTER IS DEVICE NUMBER 2
         CALL 0008        ;PRINT THE CHARACTER. IOS ENTRY
         RET
```

## 4.2.7 I/O ROUTER

### 4.2.7.1 Physical Device Identification

The physical devices are referred to by their physical location rather than their function. The following diagram indicates how a single byte is used to identify a physical device in the NPC:



```
 ---------------------------------
 | x | 1 | 1 | 1 | n | n | n | n |
 ---------------------------------
   |   |       |   |           |
   |    -------     -----------
   |       |             |
   |       |              -------------- DEVICE_NUMBER
   |       |                             indicates the particular device
   |       |                             number at a device location.
   |       |                             Device number 0 is reserved.
   |       |                             Numbering should start at 1
   |       |                             and increase sequentially.
   |       |                             A particular device number
   |       |                             can indicate either an input
   |       |                             or an output device, but NOT
   |       |                             both.
   |       |
   |        ---------------------------- DEVICE_LOCATION
   |                                     For x=0
   |                                      000  -at keyboard I/F
   |                                      001  -at TMS9918A
   |                                      010  -at Printer
   |                                      011  -at Sound Generator
   |                                      100  -at NNI I/F
   |                                     For x=1
   |                                      000  -at expansion slot 0
   |                                      001  -at expansion slot 1
   |                                      010  -at expansion slot 2
   |                                      011  -at expansion slot 3
   |
    ------------------------------------ BOARD LOCATIONS

                                    IF x=
                                         0    -on processor board
                                         1    -on expansion bus
```

Physical devices are deemed by the NPC IOS to be one of two "sexes". These are serial-oriented and packet-oriented. Serial-oriented devices are dealt with one character at a time. These are devices such as the TMS-9918A, the KEYBOARD and the PRINTER.

Packet oriented devices are dealt with a block of data at a time. Packets have a particular protocol associated with them and are generally associated with mass storage devices such as the NNI and floppy disks.

#### 4.2.7.2 Logical Device Identification

The following logical devices are defined:

| | | |
|---|---|---|
| KEYBOARD: | (input portion of CONSOLE) | 0 |
| SCREEN: | (output portion of CONSOLE) | 1 |
| LIST: | (output) | 2 |
| READER: | (input device) | 3 |
| PUNCH: | (output device) | 4 |

#### 4.2.7.3 I/O Routing Entry Point

Assignments of physical devices to logical devices are performed by using the I/O Router Entry Point. This call only allows serial-oriented physical devices to be attached to Logical devices. Mass-storage devices are handled through the Segment Loader Interface. The ATTACH entry point has the following format:

```
I/O_ROUTER: ATTACH                         (call number 8AH)
     -attaches a particular physical device or mass storage
      file to a logical device
     -entry parameters:
        C  Register: 8A Hex
        E  Register: PHYSICAL_DEVICE
        D  Register: LOGICAL_DEVICE
```

Where LOGICAL_DEVICE is the byte value of a logical device as identified in the section above and PHYSICAL_DEVICE is the byte value of a physical device, as identified above. This call will cause all subsequent I/O to the logical device to be performed by the physical device attached. This call is available in the DOS.

THIS PAGE LEFT INTENTIONALLY BLANK

**4.3 Basic Operating Software (BOS)**

This level of the operating system provides the key operating control software for the NABU P.C.. It interfaces to the I/O handlers, the Downloadable Operating Software and application programs.

BOS Routines may be linked to the applications program at run time by using the IOS DOS call number 90H (LINK_BOS_ROUTINES).

The application program is written with a jump table, with one entry in the table for each low level BOS routine accessed. Each entry is 3 bytes long. The exact structure is:

```
   TYPE
        ANENTRY:  TYPE   ARRAY[1..3] OF BYTE;

   VAR
     BOS_LINK_TABLE:     RECORD OF
                          LENGTH:             BYTE;
                          ENTRY[1..LENGTH]    ANENTRY;
                         END;
```

The exact format of the IOS DOS call is:

```
LINK_BOS_ROUTINES                             (call number 90H)
     -used to link BOS Routines to an
      application program
     -entry parameters:
        C  Register: 90 Hex
        DE Register: Pointer to a BOS_LINK_TABLE
     -is not re-entrant
```

The first byte of each entry contains the number of the BOS routine to be linked to. When the LINK_BOS_ROUTINES call is made, the IOS will go through the link table, placing the appropriate absolute jump instruction into each entry to link it to the desired routine. The application program can then jump directly through the link table to the desired routine.

Example:

```
 Before DOS Call 90H                          |  After DOS Call 90H

 LNKTAB: DB      2    ;2 entries              |  DB      2
         DB      02H  ;BOS Call - CRBEG       |  JP
         DW      0                            |         <CRBEG>
         DB      03H  ;BOS Call - CREND       |  JP
         DW      0                            |          <CREND>
```

The BOS routine numbers (HEX) are assigned as follows:

| | | | |
|---|---|---|---|
| 00 | VREGRD | 01 | VTABRD |
| 02 | CRBEG | 03 | CREND |
| 04 | VREGWR | 05 | VSTATR |
| 06 | VNAMET | 07 | VCOLRT |
| 08 | VPTRNT | 09 | VSATRT |
| 0A | VSPRST | 0B | VBLKON |
| 0C | VBLKOF | 0D | VRAMRD |
| 0E | VRAMWR | 0F | FASTL8 |
| 10 | FASTLD | 11 | FASTD8 |
| 12 | FASTDU | 13 | VRAML8 |
| 14 | VRAMLD | 15 | VRAMD8 |
| 16 | VRAMDU | 17 | SPMARK |
| 18 | SPMOVE | 19 | SPCOLR |
| 1A | SPNAME | 1B | RPATRN |
| 1C | LPATRN | 1D | CHADR |
| 1E | VFILL | 1F | XYLOC |
| 20 | PUTPAT | 21 | GETPAT |
| 22 | SETMSG | 23 | PUTMSG |
| 24 | GETMSG | 25 | VSETTX |
| 26 | VSETG1 | 27 | VSETG2 |
| 28 | VSETSP | 29 | MUL88 |
| 2A - 34 Reserved | | | |
| | | 35 | AUDRD |
| 36 | AUDWR | 37 | CLKPR |
| 38 | HOINT | 39 | CREGW |
| 3A | VMOVI | 3B | VMOVD |
| 3C | FASTRD | 3D | FASTWR |
| 3E | SETMK | | |

The BOS calls use several dedicated data structures. They are defined as follows and are referred to in the specific BOS routines.

The MESSAGE_CONTROL_BLOCK consists of :

```
X LOCATION on screen     (byte)
Y LOCATION on screeen    (byte)
LENGTH OF MESSAGE        (byte)
DATA TO BE WRITTEN       (byte(s))
```

The PATTERN_DEFINITION_TABLE consists of:

```
# OF ENTRIES IN TABLE      ;1 BYTE
BLOCK 1                    ;character 1
BLOCK 2                    ;character 2
...
BLOCK N                    ;character N
```

```
EACH BLOCK CONTAINS:
  # OF PATTERN      ;1 BYTE
  PATTERN DEF.      ;8 BYTES WICH REPRESENT THE DEFINITION.
```

## ROUTINE NAME: AUDRD

FUNCTION:
Read the audio chip

DESCRIPTION:
This routine reads from the GI complex sound generator. The register to be read is passed in C and the data is returned in A

PARAMETERS PASSED:
C Reg: Number of sound register to be read

PARAMETERS RETURNED:
A Reg: Value of sound register read

REGISTERS USED:
Flags, A, C
2 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 35 - Re-entrant

COMMENTS AND WARNINGS:
This call can be used by the application program to read the current status of the audio chip's registers. There are fourteen audio registers uses by the application. For more information on the audio chip, see Section 3.3.

RELATED ROUTINES:
AUDWR - write to the audio chip

ROUTINE NAME: AUDWR

FUNCTION:
Write to the audio chip

DESCRIPTION:
This routine writes to the GI complex sound generator used by the NABU PC. The register to be written to is passed in C and the data to be written is passed in E. The routine prevents writes to registers 0E or 0F.

PARAMETERS PASSED:
C Reg: Number of sound register to be written to
E Reg: Data to be written

PARAMETERS RETURNED:
NONE

REGISTERS CLOBBERED:
A, C, E, Flags
2 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 36 - Re-entrant

COMMENTS AND WARNINGS:
This routine writes to a specified register within the audio chip. Fourteen registers are used for sound generator. For more information on programming sound see section 3.3.

RELATED ROUTINES:
AUDRD - read from the audio chip

## ROUTINE NAME: CHADR

FUNCTION:
Return VRAM address for a particular pattern

DESCRIPTION:
This routine will return the VRAM address for a particular pattern in a pattern table. The pattern number is passed in the C register, the address returned in the HL pair. The base address of the pattern table is passed in DE.

PARAMETERS PASSED:
C = pattern number
DE = base address of PATTERN_DEF_TAB

PARAMETERS RETURNED:
HL = address of pattern

REGISTERS USED:
BC,HL
Stack use = 2 bytes

ROUTINE TYPE GLOBAL - BOS No. 1D - Re-entrant

COMMENTS AND WARNINGS:
This routine allows the application to obtain the exact address in Video RAM (VRAM), where a given character resides. It is assumed that the pattern table has already been defined and the base address is known by the application.

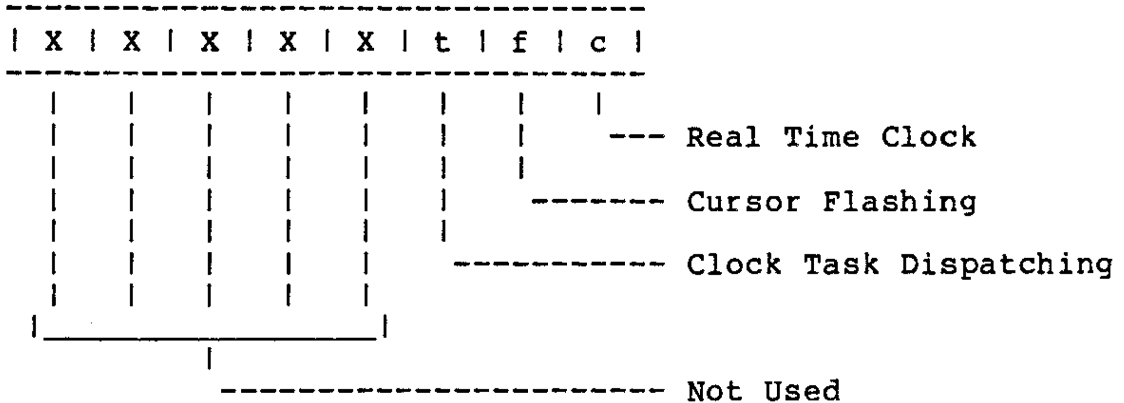
RELATED ROUTINES:
VPTRNT - set pattern table base address
VRAMLD - load Video RAM

## ROUTINE NAME: CLKPR

FUNCTION:
Change processing of real time clock functions

DESCRIPTION:
This routine is used to control processing of real time functions. Three functions may be controlled - clock user task handling, screen driver cursor flashing and real time clock updating.

These functions may be turned on or off at will by the applications program. This might be done to get more processor resources, or to get special control of these functions. Each function is controlled by a bit in a control word as shown below:




PARAMETERS PASSED:
E Reg: Data to indicate state to set
1 = process 0 = turn off
C Reg: Mask Data. Bits in E which are to actually be considered are set in the mask.

PARAMETERS RETURNED:
A Reg: New value of control word

REGISTERS USED:
A, C, Flags
4 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 37 - Re-entrant

COMMENTS AND WARNINGS:
Of course altering real time processing can cause problems. Use with caution!

## ROUTINE NAME: CRBEG

FUNCTION:
Critical region begins (disable interrupts)

DESCRIPTION:
This routine is used to delineate the beginning of a "critical region". A critical region is any section of code which, because it uses software timing, accesses data used by another task, or is not reentrant and can be called by more than one task, must run with interrupts disabled. Note that critical regions must be made as short as possible, or keyboard strokes and clock ticks may be lost.

PARAMETERS PASSED:
None

PARAMETERS RETURNED:
None

REGISTERS USED:
None
2 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 2 - Re-entrant

COMMENTS AND WARNINGS:
All interrupts are disabled by this call. Long critical regions may result in loss of clock ticks or keyboard data.

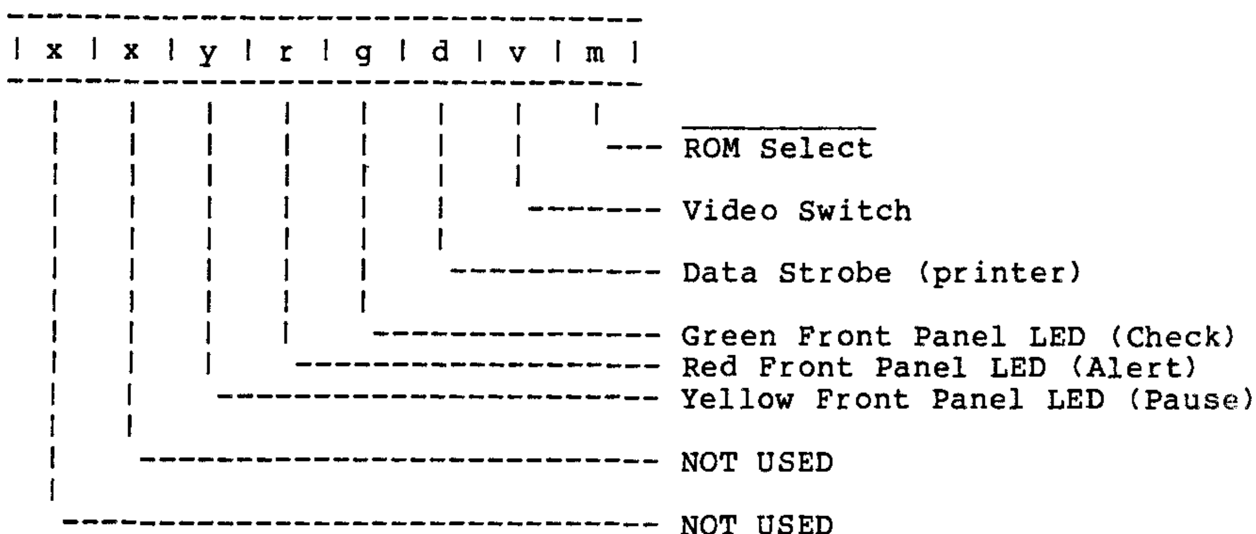
NOTE:
1. critical regions may be nested safely about 100 deep.
2. the number of CRENDs must match the number of CRBEG's

RELATED ROUTINES:
CREND - critical region ends

ROUTINE NAME: CREGW

FUNCTION:
Write to the hardware control register

DESCRIPTION:
This routine is used to write the control register port in the NABU P.C. The control register port is a write-only register with the following format:




PARAMETERS PASSED:
E Reg: Data to be Written to Port
C Reg: Mask Data. Bits in E which are to actually be written are set in the mask.

PARAMETERS RETURNED:
A Reg: New value of control register

REGISTERS USED:
A, C, Flags
4 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 39 - Re-entrant

COMMENTS AND WARNINGS:
Altering anything other than the Video Switch and the yellow and green alerting LED may cause a small disaster. Use with care.
Toggling the Video switch allows the application to switch the signal to the T.V. from the television broadcast to the video chip output and back again.

## ROUTINE NAME: CREND

FUNCTION:
Critical region ends

DESCRIPTION:
This routine is used to delineate the end of a "critical region". A critical region is any section of code which, because it uses software timing, accesses data used by another task, or is not reentrant and can be called by more than one task, must run with interrupts disabled. Note that critical regions must be made as short as possible, or keyboard strokes and clock ticks etc. may be lost. A CREND must be used to end a critical region started by a CRBEG.

PARAMETERS PASSED:
None

PARAMETERS RETURNED:
None

REGISTERS USED:
None
2 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 3 - Re-entrant

COMMENTS AND WARNINGS:
See "Critical Regions" in the section "DOS Calls - Interrupts and Tasking."

NOTE
1. critical regions may safely be nested 100 deep.
2. the number of CREND's must match the number of CRBEG's.

RELATED ROUTINES:
CRBEG - critical region begins

## ROUTINE NAME: FASTD8

FUNCTION:
Read a string of bytes from the VRAM.

DESCRIPTION:
This routine is used to read a string of bytes from the VRAM. The length of the data to be read is passed in reg BC and the memory address where the data is to be placed is passed in reg DE. The start address in VRAM is passed in HL. Since 16 bit pointers are used, anywhere from 0 to 16K of data may be transfered with this routine. The entry point FASTD8 may be used if the length of data is less than 256 bytes and the length is passed in the C reg only. This routine keeps interrupts (except keyboard interrupts) disabled for the duration of the VRAM dump. This makes the dump very fast, but susceptable to loss of clock ticks or other interrupts.

PARAMETERS PASSED:
C Reg: Length of data block to be read
DE Reg: Start of area to dump to in RAM
HL Reg: Start of source area in VRAM

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, DE, HL, Flags
6 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 11 - Re-entrant

COMMENTS AND WARNINGS:
This routine keeps interrupts (except keyboard interrupts) disabled for a long period of time. Interrupts may be lost!

RELATED ROUTINES:
FASTDU - fast dump
VRAMD8 - dump Video RAM (up to 256 bytes)
VRAMDU - dump Video RAM

## ROUTINE NAME: FASTDU

FUNCTION:
Read a string of bytes from the VRAM.

DESCRIPTION:
This routine is used to read a string of bytes from the VRAM. The length of the data to be read is passed in reg BC and the memory address where the data is to be placed is passed in reg DE. The start address in VRAM is passed in HL. Since 16 bit pointers are used, anywhere from 0 to 16K of data may be transfered with this routine. This routine keeps interrupts (except keyboard interrupts) disabled for the duration of the VRAM dump. This makes the dump very fast, but susceptable to loss of clock ticks or other interrupts.

PARAMETERS PASSED:
BC Reg: Length of data block to be read
DE Reg: Start of area to dump to in RAM
HL Reg: Start of source area in VRAM

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, DE, HL, Flags
6 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 12 - Re-entrant

COMMENTS AND WARNINGS:
This routine keeps interrupts (except keyboard interrupts) disabled for a long period of time. Interrupts may be lost!

This will affect tasks attached to the clock, and software timing if a very large number of bytes are being read.

RELATED ROUTINES:
FASTD8 - fast dump (less than 256 bytes)
VRAMD8 - Video RAM dump (less than 256 bytes)
VRAMDU - Video RAM dump

## ROUTINE NAME: FASTL8

FUNCTION:
write a string of bytes to the VRAM.

DESCRIPTION:
This routine is used to write a string of bytes to the VRAM. The length of the data to be written is passed in reg BC and the memory address of the start of the data is passed in reg DE. The start address in VRAM is passed in HL. Since 16 bit pointers are used, anywhere from 0 to 16K of data may be transfered with this routine. The entry point FASTL8 may be used if the length of data is less than 256 bytes and the length is passed in the C reg only. This routine keeps interrupts (except keyboard interrupts) disabled for the duration of the VRAM load. This makes the load very fast, but susceptable to the loss of clock ticks or other interrupts.

PARAMETERS PASSED:
C Reg: Length of data block to be written
DE Reg: Start address of data block in RAM
HL Reg: Destination of data in VRAM

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, DE, HL, Flags
6 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 0F - Re-entrant

COMMENTS AND WARNINGS:
This routine keeps interrupts (except keyboard inter rupts) disabled for a long period of time. Interrupts may be lost!

RELATED ROUTINES:
FASTLD - fast load
VRAML8 - load Video RAM (up to 256 bytes)
VRAMLD - load Video RAM

## ROUTINE NAME: FASTLD

FUNCTION:
Write a string of bytes to the VRAM.

DESCRIPTION:
This routine is used to write a string of bytes to the VRAM. The length of the data to be written is passed in reg BC and the memory address of the start of the data is passed in reg DE. The start address in VRAM is passed in HL. Since 16 bit pointers are used, anywhere from 0 to 16K of data may be transfered with this routine. This routine keeps interrupts (except keyboard interrupts) disabled for the duration of the VRAM load. This makes the load very fast, but susceptable to the loss of clock ticks or other interrupts.

PARAMETERS PASSED:
BC Reg: Length of data block to be written
DE Reg: Start address of data block in RAM
HL Reg: Destination of data in VRAM

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, DE, HL, Flags
6 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 10 - Re-entrant

COMMENTS AND WARNINGS:
This routine keeps interrupts (except keyboard inter rupts) disabled for a long period of time. Interrupts may be lost!

This will affect tasks attached to the clock, and software timing if a large number of bytes are being read.

RELATED ROUTINES:
FASTL8 - fast load (less than 256 bytes)
VRAML8 - load Video RAM (less than 256 bytes)
VRAMLD - load Video RAM

## ROUTINE NAME: ASTRD

FUNCTION:
Read a single byte of data from TMS9918A VRAM - unprotected

DESCRIPTION:
This routine is used to read a single byte of data from TMS9918A VRAM. The address to be read is passed in reg BC, the value of the VRAM at that location is returned in reg A. This routine is not protected using the CRBEG and CREND routines.

PARAMETERS PASSED:
BC Reg: Location of VRAM to be read from

PARAMETERS RETURNED:
A Reg: Contents of VRAM at Location

REGISTERS USED:
A,F
4 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 3C - Re-entrant

COMMENTS AND WARNINGS:
USE AT YOUR OWN RISK!!

RELATED ROUTINES:

| | |
|---|---|
| FASTWR | - fast write of one byte to Video RAM |
| VRAMRD | - read one byte from Video RAM |
| VRAMWR | - write one byte to Video RAM |

## ROUTINE NAME: FASTWR

FUNCTION:
Write a single byte of data to TMS9918A VRAM - unprotected

DESCRIPTION:
This routine is used to write a single byte of data from TMS9918A VRAM The address to be written is passed in reg BC. The data to be written is passed in Register E. This routine is not protected using the usual CRBEG and CREND.

PARAMETERS PASSED:
BC Reg: Location of VRAM to be written to
E Reg: Data to be written

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, flags
4 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 3D - Re-entrant

COMMENTS AND WARNINGS:
USE AT YOUR OWN RISK!!

RELATED ROUTINES:

| | |
|---|---|
| FASTRD | - fast read of one byte of Video RAM |
| VRAMRD | - read one byte from Video RAM |
| VRAMWR | - write one byte to Video RAM |

## ROUTINE NAME: GETMSG

FUNCTION:
Get message from screen

DESCRIPTION:
GETMSG gets a string of patterns from the screen. A pointer to a MESSAGE_CONTROL_BLOCK is passed in reg BC.

PARAMETERS PASSED:
BC = pointer to message control block

PARAMETERS RETURNED:
None

REGISTERS USED:
A,B,C,D,E,F,H,L
2+ Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 24 - Re-entrant

COMMENTS AND WARNINGS:
The message control block is set up in RAM by the application program.

RELATED ROUTINES:
PUTMSG - put message on the screen

## ROUTINE NAME: GETPAT

FUNCTION:
Get pattern number for any X-Y location on screen

DESCRIPTION:
GETPAT gets a pattern number from a specific X-Y location on the screen. The pattern number is returned in the A register, the X location passed in the C register, and the Y location passed in the E register.

PARAMETERS PASSED:
C = X location
E = Y location

PARAMETERS RETURNED:
A = pattern number

REGISTERS USED:
A,BC,DE,HL
Stack use = 6 bytes

ROUTINE TYPE GLOBAL - BOS No. 21 - Re-entrant

COMMENTS AND WARNINGS:
NOTE: The screen must be set up already. i.e. pattern table, sprite tables, colour table, and attribute table.

RELATED ROUTINES:
PUTPAT - put a pattern on the screen.

ROUTINE NAME: HOINT
-------------------

DESCRIPTION:
Initializes systems on the NABU PC. Calls all initialization routines for all devices and drivers, sets the control register of the NABU PC, and initializes the interrupt mask.

PARAMETERS PASSED: None.

PARAMETERS RETURNED: None.

REGISTERS CLOBBERED: ALL

ROUTINE TYPE GLOBAL - BOS No. 38 - Re-entrant

COMMENTS AND WARNINGS:
This routine is not usually needed by an application program.

## ROUTINE NAME: LPATRN

FUNCTION:
Load pattern definitions into VRAM memory

DESCRIPTION:
LPATRN loads pattern definitions into a VRAM pattern table. The patterns to be loaded are put into a PATTERN_DEFINITION_TABLE, which is described below. A pointer to the PATTERN_DEFINITION_TABLE is passed in the BC register. The base address of the pattern table is passed in DE.

PARAMETERS PASSED:
BC = pointer to PATTERN_DEF_TAB
DE = Base address of table

REGISTERS USED:
A,B,C,D,E,F,H,L
2 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 1C - Re-entrant

COMMENTS AND WARNINGS:

This routine can be used to load pattern definitions from RAM into Video RAM (VRAM). The base address of the table in VRAM to which the pattern definitions are going must already be established i.e. base address set. The pattern table, sprite definition table, and the colour table can be loaded with this routine.

RELATED ROUTINES:
RPATRN - load pattern definitions for pattern table

## ROUTINE NAME: MUL88

FUNCTION:
Multiply two eight bit numbers

DESCRIPTION:
MUL88 multiplies two 8 bit numbers together to yield a 16 bit result. The numbers to be multiplied are passed in the C and E registers, the answer is returned in both HL and BC

PARAMETERS PASSED:
C = multiplicand
E = multiplier

PARAMETERS RETURNED:
BC = result
HL = result

REGISTERS USED:
BC,DE,HL
Stack use = 0

ROUTINE TYPE GLOBAL - BOS No. 29 - Re-entrant

COMMENTS AND WARNINGS:
None

## ROUTINE NAME: PUTMSG

FUNCTION:
Put message on screen

DESCRIPTION:
PUTMSG places text on the screen. A pointer to a MESSAGE_CONTROL_BLOCK is passed in the BC registers.

PARAMETERS PASSED:
BC = pointer to message control block

PARAMETERS RETURNED:
None

REGISTERS USED:
A,B,C,D,E,F,H,L
2+ Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 23 - Re-entrant

COMMENTS AND WARNINGS:

This routine assumes that a graphics or text mode, and pattern tables are defined. It also assumes that the pattern table loaded in Video RAM has an ASCII character set loaded into the appropriate locations within the pattern table.

RELATED ROUTINES:
GETMSG - get a message from the screen

## ROUTINE NAME: PUTPAT

FUNCTION:
Put pattern at any X-Y location on screen

DESCRIPTION:
PUTPAT places any pattern definition at a specific X-Y location on the screen. The pattern number is passed in L register, the X location in the C register, and the Y location in the E register.

PARAMETERS PASSED:
C = X location on screen
E = Y location
L = pattern number

PARAMETERS RETURNED:
None

REGISTERS USED:
BC,DE,HL
Stack use = 6 bytes

ROUTINE TYPE GLOBAL - BOS No. 20 - Re-entrant

COMMENTS AND WARNINGS:

The graphics or text mode must already be defined before this routine is called. The pattern tables must also be set up (base addresses set, and pattern definitions loaded).

RELATED ROUTINES:
GETPAT - get pattern number for an X-Y screen location

ROUTINE NAME: RPATRN
--------------------

FUNCTION:
Load pattern definitions into screen pattern table

DESCRIPTION:
RPATRN loads pattern definitions into the screen's pattern table. A pointer to a PATTERN_DEFINITION_TABLE is passed in register BC. The PATTERN_TABLE address is assumed to be at VPTRNAD.

PARAMETERS PASSED:
BC reg = pointer to PATTERN_DEF_TAB

PARAMETERS RETURNED:
None

REGISTERS USED:
A,B,C,D,E,F,H,L
2+ Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 1B - Re-entrant

COMMENTS AND WARNINGS:
It is assumed the base address of the pattern table (VPTRNAD) has already been set.
VPTRNAD is defined and set using BOS routine VPTRNST.

RELATED ROUTINES:
LPATRN - load pattern definitions into Video RAM

## ROUTINE NAME: SETMSG

FUNCTION:
Set up screen message

DESCRIPTION:
SETMSG sets up the VDP and all parameters according to a MESSAGE_CONTROL_BLOCK. The user may then load or dump to VRAM, and the patterns will be placed appropriately. The pointer to the message control block is passed in the BC register pair. The user should use VRAML8 or VRAMD8 immediatley after this.

THE A REGISTER CONTAINS THE TYPE OF SETMSG. 0 = FOR READ
1 = FOR WRITE

PARAMETERS PASSED:
BC = pointer to MESSAGE_CONTROL_BLOCK

PARAMETERS RETURNED:
C = Length of message
DE = Pointer to data to be read/displayed.
HL = VRAM address to read/write.

REGISTERS USED:
A,BC,DE,HL
Stack use = 6 bytes

ROUTINE TYPE GLOBAL - BOS No. 22 - Re-entrant

COMMENTS AND WARNINGS:

The routine PUTMSG is made up of SETMSG and VRAML8. SETMSG should be used by the application program for dumping VRAM contents into RAM.

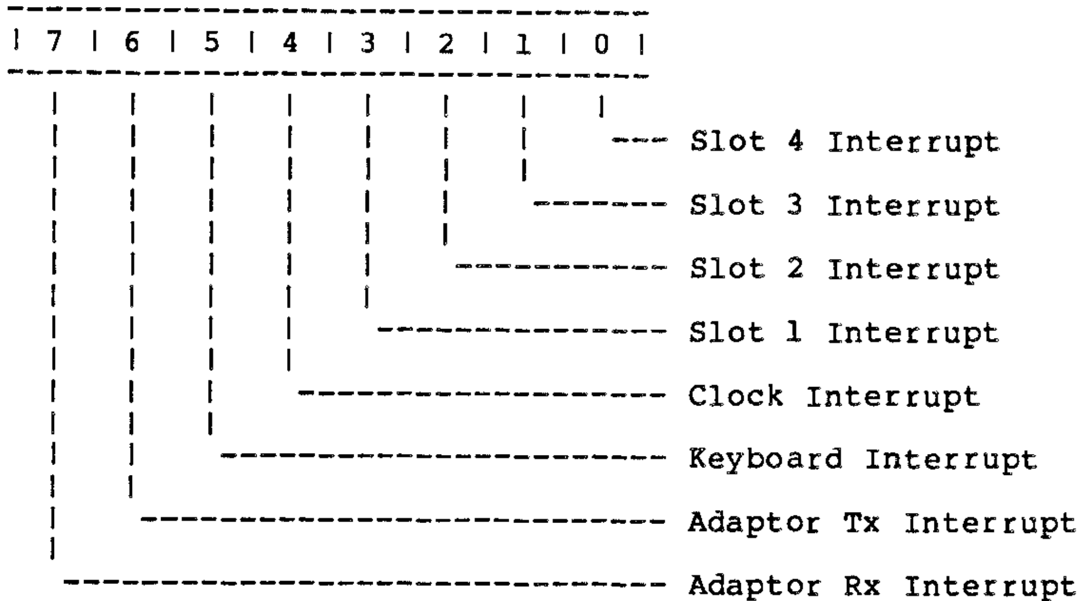
RELATED ROUTINES:

| | |
|---|---|
| PUTMSG | - put message on the screen |
| VRAML8 | - load up to 256 bytes into Video RAM |
| VRAMD8 | - dump up to 256 bytes into RAM |

## ROUTINE NAME: SETMSK

FUNCTION:
Write hardware interrupt control register and mask.

DESCRIPTION:
This routine is used to write or set the interrupt control register port in the NABU PC. The control register is a write-only register with the following bit format:




PARAMETERS PASSED:
E Reg: Data to be written to the port.
C Reg: Mask Data. Bits in E that are to actually be written are set in the mask.

PARAMETERS RETURNED:
A Reg: Previous value of the control register.

REGISTERS USED:
A, C, Flags
4+ Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 3EH - Non Re-entrant

COMMENTS AND WARNINGS:
The implications of playing with the interrupt control register are considerable. Use with caution.

## ROUTINE NAME: SPCOLR

FUNCTION:
Set the colour of a sprite.

DESCRIPTION:
This routine is used to set the colour of a sprite. The sprite number is passed in register C and the new sprite colour is passed in register E.

PARAMETERS PASSED:
C Reg: Number of sprite to change colour of
E Reg: Number of new colour

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, DE, HL, Flags
6 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 19 - Re-entrant

COMMENTS AND WARNINGS:

NOTE that the colour, the sprite location (SPMOVE) and the sprite pattern (SPNAME) must all be called before a sprite appears on the screen.

RELATED ROUTINES:

| | |
|---|---|
| SPMOVE | - move sprite |
| SPNAME | - assign pattern definition to sprite |
| SPMARK | - mark last sprite |

## *ROUTINE NAME:* SPMARK

FUNCTION:

*Mark the end of a sprite attribute table*

DESCRIPTION:

This routine is used to mark the end of a sprite attribute table. The number of the sprite to be marked (ie. the sprite AFTER the last sprite) is passed in the C register.

PARAMETERS PASSED:

C Reg: Number of sprite to be marked

PARAMETERS RETURNED:

NONE

REGISTERS USED:

A, BC, DE, HL, Flags
*6 Bytes of stack used*

ROUTINE TYPE GLOBAL - BOS No. 17 - Re-entrant

COMMENTS AND WARNINGS:

NOTE: If a sprite pattern is defined on the sprite number that was marked by this routine, the sprite mark is effectively removed.

RELATED ROUTINES:

SPMOVE - move sprite
SPCOLR - set sprite colour
SPNAME - assign pattern definition to sprite

## ROUTINE NAME: SPMOVE

FUNCTION:
Move a sprite on the display.

DESCRIPTION:
This routine is used to move a sprite on the display. The new X location is passed in L, the new Y location is passed in E and the number of the sprite to be moved is passed in register C.

PARAMETERS PASSED:
C Reg: Number of sprite to be moved
E Reg: New Y location
L Reg: New X location

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, DE, HL, Flags
6 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 18 - Re-entrant

COMMENTS AND WARNINGS:
This routine is also used to define the first location of a sprite.
NOTE: The colour, and pattern must also be defined to have the sprite appear on the screen.

RELATED ROUTINES:
SPMARK - mark the last sprite being used
SPCOLR - set sprite colour
SPNAME - set sprite pattern definition

## ROUTINE NAME: SPNAME

FUNCTION:
Set the pattern name associated with a sprite.

DESCRIPTION:
This routine is used to set the pattern name associated with a sprite. The sprite number is passed in register C and the new sprite pattern name is passed in register E.

PARAMETERS PASSED:
C Reg: Number of sprite to change pattern of
E Reg: Number of new pattern

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, DE, HL, Flags
6 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 1A - Re-entrant

COMMENTS AND WARNINGS:
NOTE:
1. The pattern name is the pattern which resides in the sprite pattern table in Video RAM.
2. The colour and the location of the sprite must be defined before the sprite will appear on the screen.

RELATED ROUTINES:

| | |
|---|---|
| SPMARK | - mark the last sprite being used |
| SPCOLR | - set the colour of the sprite |
| SPMOVE | - set the location of the sprite |

## ROUTINE NAME: VBLKOF

FUNCTION:
Unblanks (turns on) the TMS9918A video display.

DESCRIPTION:
This routine unblanks (turns on) the TMS9918A video display. It requires no parameters.

PARAMETERS PASSED:
None

PARAMETERS RETURNED:
None

REGISTERS USED:
A, BC, E, HL, Flags
4 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 0C - Re-entrant

COMMENTS AND WARNINGS:
The definition of the screen, i.e. mode, patterns etc, should be done before unblanking the screen. When a mode is selected be it TEXT, GRAPHICS 1, or GRAPHICS 2, the screen is "blanked" and remains blank until the VBLKOF routine "unblanks" it.

RELATED ROUTINES:

| | |
|---|---|
| VBLKON | - blank the video display |
| VSETTX | - set to TEXT mode |
| VSETG1 | - set to GRAPHICS 1 mode |
| VSETG2 | - set to GRAPHICS 2 mode |

ROUTINE NAME: VBLKON

FUNCTION:
Blanks the TMS9918A video display.

DESCRIPTION:
This routine blanks the TMS9918A video display. It requires no parameters. Blanking means all foreground colours and sprites disappear from the screen. The background colour remains.

PARAMETERS PASSED:
None

PARAMETERS RETURNED:
None

REGISTERS USED:
A, BC, E, HL, Flags
4 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 0B - Re-entrant

COMMENTS AND WARNINGS:
NOTE: The TV screen goes blank on calling this routine, but the definitions that have been set up in Video RAM remain. To regain the image on the screen, use VBLKOF.

RELATED ROUTINES:
VBLKOF - unblanks the video display

## ROUTINE NAME: VCOLRT

FUNCTION:
Set the colour table address in the TMS9918A.

DESCRIPTION:
This routine is used to set the colour table address in the TMS9918A. The full colour table address is passed in reg BC. This routine correctly writes the address into the 9918 reg 3 and stores the full colour table address in VCOLRAD for use by other routines. This routine works in Graphics II Mode by setting all the most significant bits as required by the VDP

PARAMETERS PASSED:
BC Reg: Base Address of COLOUR Table

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, E, HL, Flags
4 Bytes of Stack

ROUTINE TYPE GLOBAL - BOS No. 7 - Re-entrant

COMMENTS AND WARNINGS:
The colour table must be set up for programs using GRAPHICS 1 or GRAPHICS 2.

## ROUTINE NAME: VFILL

FUNCTION:
Fill a block of Video RAM with one character

DESCRIPTION:
This routine will fill any contiguous portion of VRAM with a particular value. The value to fill with is passed in the E register, the length to fill is passed in the BC pair. The Address in VRAM is passed in HL

PARAMETERS PASSED:
BC = length to fill
E = value to fill with
HL = address in VRAM to start

PARAMETERS PASSED:
None

PARAMETERS RETURNED:
None

REGISTERS USED:
A,B,C,D,E,F,H,L
4+ Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 1E - Re-entrant

COMMENTS AND WARNINGS:
This allows the application program, which resides in RAM, to keep from having to define large tables in RAM containing the same entry over and over again, and then copying the table into Video RAM.
This routine can be used to pad out pattern tables with the number for blanks or fill colour tables with one combination of colours.

## ROUTINE NAME: VMOVD

**FUNCTION:**

Quickly move data from one location in VRAM to another.

**DESCRIPTION:**

This routine will quickly move data from one location in VRAM to another. The data area must be less than 255 bytes long. The move is made by starting at the locations specified and moving DOWN in VRAM

**PARAMETERS PASSED:**

C Reg: Amount of data to be moved in bytes
DE Reg: End Address where data is located
HL Reg: End Address where data is to be moved to

**PARAMETERS RETURNED:**

DE Reg: One before the beginning of the source data area
HL Reg: One before the beginning of the destination data area

**REGISTERS USED:**

A, BC, DE, HL, Flags
4 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 3B - Re-entrant

**COMMENTS AND WARNINGS:**

Calling this routine with C reg equal to zero will cause 256 bytes of data to be transferred. This routine disables interrupts for the full data transfer. This may cause interrupts to be lost.

NOTE: If the value in HL is greater than the value in DE minus the value of C, then the difference will be the number of bytes "clobbered" at the start of the block of data being moved.

**RELATED ROUTINES:**

VMOVI - move data in Video RAM (incrementing from given address)

ROUTINE NAME: VMOVI
---

FUNCTION:
Quickly move data from one location in VRAM to another.

DESCRIPTION:
This routine will quickly move data from one location in VRAM to another. The data area must be less than 255 bytes long. The move is made by starting at the locations specified and moving UP in VRAM

PARAMETERS PASSED:
C Reg: Amount of data to be moved in bytes
DE Reg: Start Address where data is located
HL Reg: Start Address where data is to be moved to

PARAMETERS RETURNED:
DE Reg: One past the end of the source data area
HL Reg: One past the end of the destination data area

REGISTERS USED:
A, BC, DE, HL, Flags
4 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 3A - Re-entrant

COMMENTS AND WARNINGS:
Calling this routine with C reg equal to zero will cause 256 bytes of data to be transferred. This routine disables interrupts for the full data transfer. This may cause interrupts to be lost.
NOTE: If the value in HL is less than the value in DE plus the value of C, then the difference will be the number of bytes "clobbered" at the end of the block of data being moved.

RELATED ROUTINES:
VMOVD - move data in VRAM (decrementing from given address)

ROUTINE NAME: VNAMET

---

FUNCTION:
Set the pattern name address of the TMS9918A.

DESCRIPTION:
This routine is used to set the pattern name address of the TMS9918A. The full pattern name address is passed in reg BC. This routine correctly writes the address into the 9918 reg 2 and stores the full pattern name address in VNAMEAD for use by other routines.

PARAMETERS PASSED:
BC Reg: 16 bit base address of NAME Table

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, E, HL, Flags
4 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 6 - Re-entrant

COMMENTS AND WARNINGS:
The mode that the application is working in should already be set, i.e. TEXT or GRAPHICS 1 or GRAPHICS 2. The address should be set in accordance to the mode chosen.

RELATED ROUTINES:

VCOLRT - set colour table base address
VPTRNT - set pattern table base address
VSATRT - set sprite attribute table base address
VSPRST - set sprite pattern table base address

## ROUTINE NAME: VPTRNT

FUNCTION:
Set the pattern table address in the TMS9918A.

DESCRIPTION:
This routine is used to set the pattern table address in the TMS9918A. The full pattern table address is passed in reg BC. This routine correctly writes the address into the 9918 reg 4 and stores the full pattern table address in VPTRNAD for use by other routines. This routine works correctly in GRAPHICS Mode II by setting all the most significant bits to 1.

PARAMETERS PASSED:
BC Reg: Base Address of PATTERN Table

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, E, HL, Flags
4 Bytes of Stack

ROUTINE TYPE GLOBAL - BOS No. 8 - Re-entrant

COMMENTS AND WARNINGS:
The mode that the application is working in should already be set, i.e. TEXT or GRAPHICS 1 or GRAPHICS 2. The address should be set in accordance to the mode chosen.

RELATED ROUTINES:

| | |
|---|---|
| VCOLRT | - set colour table base address |
| VNAMET | - set name table base address |
| VSATRT | - set sprite attribute table base address |
| VSPRST | - set sprite pattern table base address |

## ROUTINE NAME: VRAMD8

FUNCTION:
Dump a string of bytes from the VRAM.

DESCRIPTION:
This routine is functionally the same as FASTD8 but are safe in an interrupt environment (and also take longer). This routine is used to dump a string of bytes from the VRAM. The length of the data to be dumped is passed in reg C and the memory address, in RAM, of the destination of the data is passed in reg DE. The start address in VRAM is passed in HL. This routine can be used on strings up to 256 bytes in length.

PARAMETERS PASSED:
C Reg: Length of data block to be dumped
DE Reg: Start of destination area in RAM
HL Reg: Start of source area in VRAM

PARAMETERS RETURNED:
HL Reg: Points one byte past end of source area in VRAM
(Useful for "Chaining" Calls)

REGISTERS USED:
A, BC, DE, HL, Flags
6 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 15 - Re-entrant

COMMENTS AND WARNINGS:
This routine is safe for use in an interrupt environment. If more than 256 bytes are to be moved use the routine VRAMDU.

RELATED ROUTINES:
VRAMDU - Video RAM dump
FASTD8 - Fast Video RAM dump (less than 256 bytes)
FASTDU - Fast Video RAM dump

ROUTINE NAME: VRAMDU

FUNCTION:
Dump a string of bytes from the VRAM.

DESCRIPTION:
This routine is functionally the same as FASTDU, but are safe in an interrupt environment (and also take longer). This routine is used to dump a string of bytes from the VRAM. The length of the data to be dumped is passed in reg BC and the memory address of the destination of the data is passed in reg DE. The start address in VRAM is passed in HL. Since 16 bit pointers are used, anywhere from 0 to 16K of data may be transfered with this routine.

PARAMETERS PASSED:
BC Reg: Length of data block to be dumped
DE Reg: Start of destination area in RAM
HL Reg: Start of source area in VRAM

PARAMETERS RETURNED:
HL Reg: Points one byte past end of source area in VRAM.
(Useful for "Chaining" Calls)

REGISTERS USED:
A, BC, DE, HL, Flags
6 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 16 - Re-entrant

COMMENTS AND WARNINGS:
This routine is safe for use in an interrupt environment. If a small string (less than 256 bytes) is to be dumped, use the routine VRAMD8

RELATED ROUTINES:

| | |
|---|---|
| VRAMD8 | - Video RAM dump (less than 256 bytes) |
| FASTD8 | - Fast Video RAM dump (less than 256 bytes) |
| FASTDU | - Fast Video RAM dump |

## ROUTINE NAME: VRAML8

FUNCTION:
Write a string of bytes to the VRAM.

DESCRIPTION:
This routine is functionally the same as FASTL8 but are safe in an interrupt environment (and also take longer). This routine is used to write a string of bytes to the VRAM. The length of the data to be written is passed in reg C and the memory address of the source of the data is DE. The start address in VRAM is passed in reg HL.

PARAMETERS PASSED:
C Reg: Length of data block to be read
DE Reg: Start of source area in RAM
HL Reg: Start of destination area in VRAM

PARAMETERS RETURNED:
HL Reg: Points one byte past end of destination area in VRAM
(Useful for "Chaining" Calls)

REGISTERS USED:
A, BC, DE, HL, Flags
6 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 13 - Re-entrant

COMMENTS AND WARNINGS:
This routine is safe for use in an interrupt environment. If more than 256 bytes of data must be loaded into Video RAM, use the routine VRAMLD.

RELATED ROUTINES:
VRAMLD - load Video RAM
FASTL8 - quick load of Video RAM (less than 256 bytes)
FASTLD - quick load of Video RAM

## ROUTINE NAME: VRAMLD

FUNCTION:
Write a string of bytes to the VRAM.

DESCRIPTION:
This routine is functionally the same as FASTLD, but are safe in an interrupt environment (and also take longer). This routine is used to write a string of bytes to the VRAM. The length of the data to be written is passed in reg BC and the memory address of the source of the data is passed in reg DE. The start address in VRAM is passed in HL. Since 16 bit pointers are used, anywhere from 0 to 16K of data may be transfered with this routine.

PARAMETERS PASSED:
BC Reg: Length of data block to be read
DE Reg: Start of source area in RAM
HL Reg: Start of destination area in VRAM

PARAMETERS RETURNED:
HL Reg: Points one byte past end of destination area in VRAM
(Useful for "Chaining" Calls)

REGISTERS USED:
A, BC, DE, HL, Flags
6 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 14 - Re-entrant

COMMENTS AND WARNINGS:
This routine is safe for use in an interrupt environment. If a small string (less than 256 bytes) is to be loaded use VRAML8.

RELATED ROUTINES:
VRAMD8 - Video RAM dump (less than 256 bytes)
FASTD8 - Fast Video RAM dump (less than 256 bytes)
FASTDU - Fast Video RAM dump

## ROUTINE NAME: VRAMRD

FUNCTION:
Read a single byte of data from TMS9918A VRAM

DESCRIPTION:
This routine is used to read a single byte of data from TMS9918A VRAM. The address to be read is passed in reg BC, the value of the VRAM at that location is returned in reg A.

PARAMETERS PASSED:
BC Reg: Location of VRAM to be read from

PARAMETERS RETURNED:
A Reg: Contents of VRAM at Location

REGISTERS USED:
A,F
4 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 0D - Re-entrant

COMMENTS AND WARNINGS:
None

RELATED ROUTINES:

| | |
|---|---|
| VRAMWR | - write one byte to Video RAM |
| FASTRD | - quick read of one byte to Video RAM |
| FASTWR | - quick write of one byte to Video RAM |

## ROUTINE NAME: VRAMWR

FUNCTION:
Write a single byte of data to TMS9918A VRAM

DESCRIPTION:
This routine is used to write a single byte of data from TMS9918A VRAM The address to be written is passed in reg BC. The data to be written is passed in Register E.

PARAMETERS PASSED:
BC Reg: Location of VRAM to be written to
E Reg: Data to be written

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, flags
4 bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 0E - Re-entrant

COMMENTS AND WARNINGS:
None

RELATED ROUTINES:
VRAMRD - read one byte of Video RAM
FASTRD - read one byte of Video RAM ... fast
FASTWR - write one byte of Video RAM ... fast

## ROUTINE NAME: VREGRD

FUNCTION:
Reads the TMS9918A video display register

DESCRIPTION:
This routine reads the TMS9918A video display register values, which are stored in RAM images. The register number to be written (0 to 7) is passed in reg C.

The following data may also be read:
8: VDP Status Register RAM Image (Updated Each Clock Interrupt
9: Current VDP Mode: 0 -text, 1 -Graphics I, 2 -Graphics II
A: Current Screen Width in Characters

The data is returned in register A

PARAMETERS PASSED:
C Reg: Register Number to be read

PARAMETERS RETURNED:
A Reg: Value of Register

REGISTERS USED:
A, BC, HL, Flags
0 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 0 - Re-entrant

COMMENTS AND WARNINGS:
The actual control registers of the video chip are write only. The IOS maintains an image of these registers allowing the application to "read" the values that are currently in the registers.

RELATED ROUTINES:
VREGWR - write to a register in the video chip

## ROUTINE NAME: VREGWR

FUNCTION:
Writes the TMS9918A video display registers.

DESCRIPTION:
This routine writes the TMS9918A video display registers. The register number to be written (0 to 7) is passed in reg C and the data to be written is passed in reg E. Note that since the TMS9918A registers are write-only, images of the registers are kept in global memory where they may be read if required.

PARAMETERS PASSED:
C Reg: Register Number to be written
E Reg: Data to be written into register

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, E, HL, Flags
4 Bytes of Stack Used

ROUTINE TYPE GLOBAL - BOS No. 4 - Re-entrant

COMMENTS AND WARNINGS:
None

RELATED ROUTINES:
VREGRD - read a register in the Video chip

## ROUTINE NAME: VSATRT

FUNCTION:
Set the sprite attributes table address in the TMS9918A.

DESCRIPTION:
This routine is used to set the sprite attributes table address in the TMS9918A. The full sprite attributes table address is passed in reg BC. This routine correctly writes the address into the TMS9918A reg 5 and stores the full sprite attributes table address in VATRIAD for use by other routines.

PARAMETERS PASSED:
BC Reg: Base Address of Sprite ATTRIBUTES table

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, E, HL, Flags
4 Bytes of Stack

ROUTINE TYPE GLOBAL - BOS No. 9 - Re-entrant

COMMENTS AND WARNINGS:
This routine must be called when setting up the video for GRAPHICS 1 or GRAPHICS 2 mode.

## ROUTINE NAME: VSETG1

FUNCTION:
Set video for graphics 1 mode

DESCRIPTION:
VSETG1 sets the VDP for graphics 1 mode, blanked display, 16*16 sprites, 1X magnification. The user must use the VBLKOFF routine to enable the display.

PARAMETERS PASSED:
None

PARAMETERS RETURNED:
None

REGISTERS USED:
A,B,C,D,E,F
2 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 26 - Re-entrant

COMMENTS AND WARNINGS:
This routine does not set base addresses of tables nor does it load pattern sets into Video RAM.

RELATED ROUTINES:
VSETG2 - set to GRAPHICS 2 mode
VSETTX - set to TEXT mode

## ROUTINE NAME: VSETG2

FUNCTION:
Set video for graphics 2 mode

DESCRIPTION:
VSETG2 sets the VDP for graphics 2 mode, blanked display, 16*16 sprites, 1X magnification. The user must use the VBLKOFF routine to enable the display.

PARAMETERS PASSED:
None

PARAMETERS RETURNED:
None

REGISTERS USED:
A,B,C,D,E,F
2 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 27 - Re-entrant

COMMENTS AND WARNINGS:
This routine does not set base addresses of tables nor does it load pattern sets into Video RAM.

RELATED ROUTINES:
VSETG1 - set to GRAPHICS 1 mode
VSETTX - set to TEXT mode

## ROUTINE NAME: VSETSP

FUNCTION:
Set sprite size and magnification

DESCRIPTION:
VSETSPA sets the sprite size and magnification. The sprite size is passed in the C register (0=8*8, 1=16*16). The sprite magnification is passed in the E register (0=1X, 1=2X). The user must first set the mode using one of the above three routines.

PARAMETERS PASSED:
C = sprite size (0 = 8*8,1 = 16*16)
E = sprite magnification (0 =1x,1 =2x)

PARAMETERS RETURNED:
None

REGISTERS USED:
A,B,C,D,E,F,H,L
2+ Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 28 - Re-entrant

COMMENTS AND WARNINGS:
Defaults for sprite size and magnification are set when the mode (TEXT, GRAPHICS 1, or GRAPHICS 2) is set

## ROUTINE NAME: VSETTX

FUNCTION:
Set video for text mode

DESCRIPTION:
VSETTXT sets the VDP for text mode, blanked display, 16*16 sprites, 1X magnification. Please NOTE that sprites will NOT appear in text mode even though the video chips' registers are set up for sprites. The user must use the VBLKOFF routine to enable the display.

PARAMETERS PASSED:
None

PARAMETERS RETURNED:
None

REGISTERS USED:
A,B,C,D,E,F
2 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 25 - Re-entrant

COMMENTS AND WARNINGS:
Sprites can not be used in text mode, however when setting the VDP register, sprite information must be provided.
This routine does not set base addresses of tables nor does it load pattern sets into Video RAM.

RELATED ROUTINES:
VSETG1 - set to GRAPHICS 1 mode
VSETG2 - set to GRAPHICS 2 mode

## ROUTINE NAME: VSPRST

FUNCTION:
Set the sprite table address in the TMS9918A.

DESCRIPTION:
This routine is used to set the sprite table address in the TMS9918A. The full sprite table address is passed in reg BC. This routine correctly writes the adddress into the TMS9918A reg 6 and stores the full sprite table address into VSPRIAD for use by other routines.

PARAMETERS PASSED:
BC Reg: Base Address of Sprite PATTERN table

PARAMETERS RETURNED:
NONE

REGISTERS USED:
A, BC, E, HL, Flags
4 Bytes of Stack

ROUTINE TYPE GLOBAL - BOS No. 0A - Re-entrant

COMMENTS AND WARNINGS:
The mode in which the video chip is to work should already be set.

RELATED ROUTINES:

| | |
|---|---|
| VNAMET | - set the name table base address |
| VCOLRT | - set the colour table base address |
| VPTRNT | - set the pattern table base address |
| VSATRT | - set the sprite attribute table base address |
| VSPRST | - set the sprite definition table base address |

## ROUTINE NAME: VSTATR

FUNCTION:
Reads the status register of the TMS9918A

DESCRIPTION:
This routine reads the status register of the TMS9918A and returns the register contents in reg A. The status register image VSTATUS is also updated. This routine may cause clock interrupts to be lost as it executes, since it will reset any pending interrupt.

PARAMETERS PASSED:
NONE

PARAMETERS RETURNED:
A Reg: VDP Status Byte

REGISTERS USED:
A,B,HL,C,F
2+ Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 5 - Re-entrant

COMMENTS AND WARNINGS:
This is a dangerous call! It may cause Clock Interrupts to be lost. Unless absolutly necessary, it is best to get the VDP Status by reading VSTATUS using VREGRD. VSTATUS will be updated every 16 msec. by the clock ISR.

## ROUTINE NAME: VTABRD

FUNCTION:
Reads the current table base address pointers

DESCRIPTION:
This routine reads the current table base address pointers in the VDP from a RAM image area. The number of the pointer to be read is passed in register C. These numbers are as follows:

0: VNAMEAD -Name Table Base Address
1: VCOLRAD -Colour Table Base Address
2: VPTRNAD -Pattern Table Base Address
3: VATRIAD -Sprite Attribute Table Base Address
4: VSPRIAD -Sprite Pattern Table Base Address

PARAMETERS PASSED:
C Reg: Register Number to be read

PARAMETERS RETURNED:
HL Reg: Register value

REGISTERS USED:
A, BC, DE, HL, Flags
0 Bytes of stack used

ROUTINE TYPE GLOBAL - BOS No. 1 - Re-entrant

COMMENTS AND WARNINGS:
None

RELATED ROUTINES:
VNAMET - set the name table base address
VCOLRT - set the colour table base address
VPTRNT - set the pattern table base address
VSATRT - set the sprite attribute table base address
VSPRST - set the sprite definition table base address

## ROUTINE NAME: **XYLOC**

FUNCTION:
Return name table address for any X-Y location on screen

DESCRIPTION:
XYLOC returns the name table address in VRAM for any X-Y location on the screen. The X location is passed in the C register, the Y location in the E register. The VRAM address is returned in both HL and BC.

PARAMETERS PASSED:
C = X location (column) on screen
E = Y location (row)

PARAMETERS RETURNED:
BC = address in VRAM
HL = address in VRAM

REGISTERS USED:
BC,DE,HL
Stack use = 4 bytes

ROUTINE TYPE GLOBAL - BOS No. 1F - Re-entrant

COMMENTS AND WARNINGS:
NOTE: The screen is 32x24 patterns in the GRAPHICS modes and 40x24 patterns in TEXT mode. If a standard T.V. set is being used, the first and last column of patterns may fall just outside the screen.

THIS PAGE LEFT INTENTIONALLY BLANK

# 5.0 EXTENDED IOS

## 5.1 Introduction

The IOS is divided into 2 sections. The Kernel contains the minimum set of IOS functions, while the Extended IOS (XIOS) contains the varied extensions. This structuring of the IOS is done to leave as much programming space as possible for applications, while not reducing or limiting the functionality of the IOS. As new features, such as I/O drivers for the varied option boards, are added to the IOS, they will be placed into the XIOS section thereby keeping the Kernel size to a minimum.

The XIOS system is located in 16 different segments on the wheel. These segments represent 16 tiering levels for billing purposes - ie. the user of the application must be authorized for the particular segment required by the application program. Each segment is divided into a number of modules ( 0 through 15 ). Each module contains a related set of functions. XIOS modules are loaded by the application by specifying which segment the module is found in (0 -> 15) and then within that segment which XIOS module is desired (again a number between 0 and 15). As they are loaded, these modules will be relocated immediately below the IOS Kernel. When an application no longer needs a module, it may delete or unload that module. The Kernel software is responsible for tracking which modules are currently operative and which ones are not.

Locations 6 and 7 will always point to the base of the total IOS, Kernel plus Extended. This will enable applications to know how much space is available. Applications normally place the stack based on the value of locations 6 and 7.

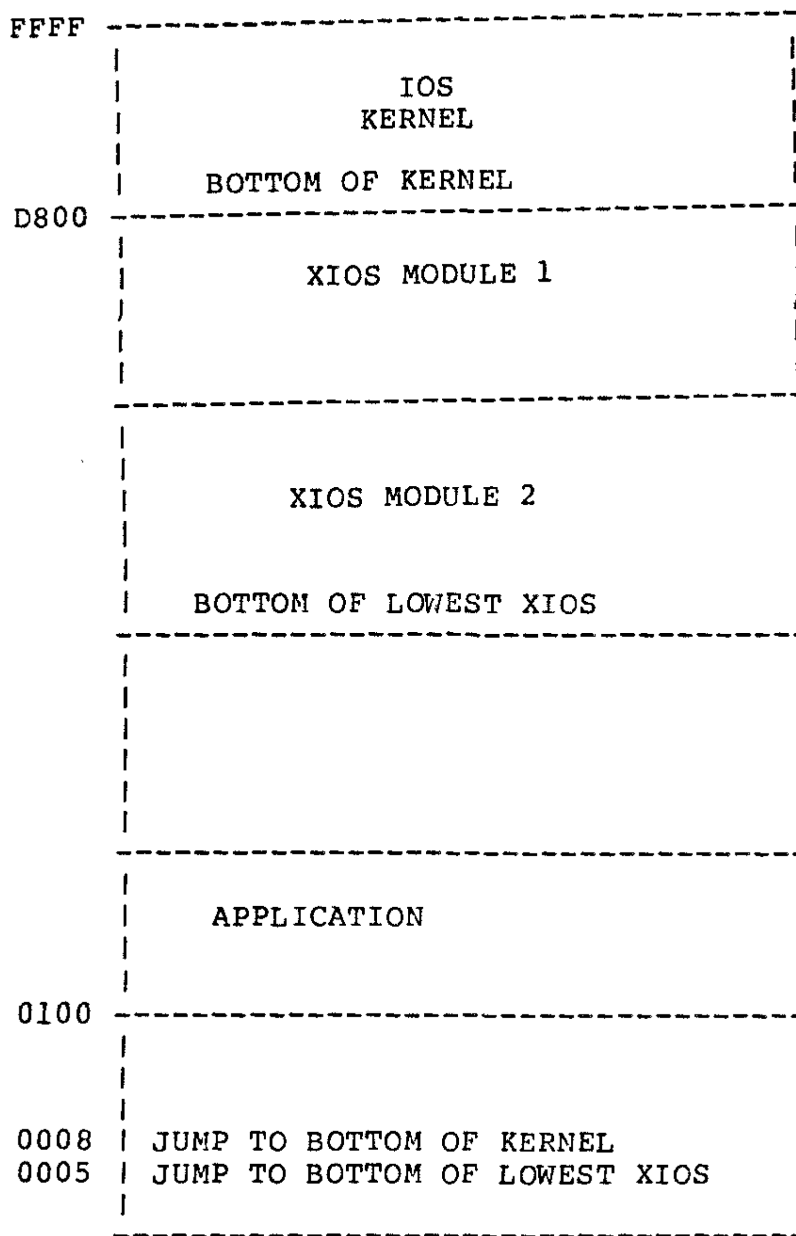
**Care must be taken that the stack is not overwritten when loading XIOS modules. Ensure that your stack is not at the top of user memory when requesting an XIOS module - that is where the XIOS module will load to.**

## 5.2 Extended IOS Module Handler

The XIOS module handler is responsible for loading, unloading, linking and keeping track of XIOS modules. This handler is included in the IOS Kernel.

### 5.2.1 Memory Structure for Loaded XIOS Modules

The structure for memory allocation of XIOS modules is depicted in the following diagram:




STRUCTURE OF IOS WITH XIOS MODULES PRESENT

### 5.2.2 Loading XIOS Modules

XIOS modules will be loaded in one module at a time. The module will be loaded in and relocated next to the very bottom of the current IOS. Locations 6 and 7 will be amended to reflect a bottom for IOS. Should difficulties occur in loading and initializing the XIOS module, the returned error code will indicate why failure occurred.

The command to load an XIOS module is a DOS command with the following format:

LOAD_XIOS_MODULE (call number 96H)

Function: Load one XIOS module

Entry Parameters: Register C = 96H

Register E = XIOS Module ID
Where XIOS Module ID is one of:
- 00 - Basic BDOS and BIOS
- 01 - Basic BDOS, Extended BDOS and BIOS
- 13 - Multi-Window Screen Driver
- 14 - 80 Column Screen Driver

Exit Parameters: Register A = Status
where Status is one of:
- 00 - Load was successful

Segment handler error codes:
- -1 - XIOS Module was not loaded because tier is not authorized
- -2 - XIOS Module was not loaded because segment buffer overflowed
- -3 - XIOS Module was not loaded because adaptor did not respond
- -4 - XIOS Module was not loaded because an incorrectly formated packet was received
- -5 - XIOS Module was not loaded because an undetermined communications protocol error occurred
- -6 - XIOS module was not loaded because it was not located in the segment

XIOS Module error codes:
Codes -10H to -70H are reserved for XIOS Module to return after initializing. These codes will be described in detail in each respective XIOS Section.

Each module is described in detail as to function and the type of support it needs with regard to hardware and other XIOS modules, in later sections of the APG.

### 5.2.3 Unloading XIOS Modules

XIOS modules will be unloaded or deleted one module at a time. Only the module at the very bottom of the IOS can be unloaded. When this happens, locations 6 & 7 will be amended to reflect the new bottom for the IOS, and indicate that memory has been freed up. In order to unload all XIOS modules, the application must unload them one at a time, until the return code indicates that no module was unloaded.

Note that when the application is terminated normally by a jump to location 0 or via the EXIT key on the keyboard (an IOS function), all resident XIOS modules are unloaded by the IOS re-boot code. This ensures that any hardware that may be "attached" to an XIOS module (eg. disk drives) is properly de-initialized (eg. drive motor is turned off).

The command to unload an XIOS module is a DOS command with the following format:

```
UNLOAD_XIOS_MODULE                                 (call number 97H)

Function: Unload one XIOS module

Entry Parameters:  Register C = 97H

Exit Parameters: Register A = Status
                    where Status is one of:
                         00 - Unload was successful and
                              unloaded module number is found
                              in register L
                         -1 - There was no XIOS module to unload
                    XIOS Module error codes:
                         Codes -10H to -70H are reserved for
                         XIOS Module to return after
                         de-initializing.  These codes will be
                         described in detail in each respective
                         XIOS Section.

                   Register L = XIOS Module ID
                    where XIOS Module ID is one of:
                         00 - Basic BDOS and BIOS
                         01 - Basic BDOS, Extended BDOS and BIOS
                         13 - Multi-Window Screen Driver
                         14 - 80 Column Screen Driver
```

Each module is described in detail as to function and the type of support it needs with regard to hardware and other XIOS modules, in later sections of the APG.

### 5.2.4 Resolving References in XIOS Modules

Different XIOS modules will require access to data structures and subroutines contained within other XIOS modules or within the Kernel.

DOS call number 99H provides the mechanism for resolving references. This call returns the address of the global variable requested. All XIOS modules containing global variables or subroutines must trap and execute this DOS call. Each global variable must be given a unique reference number. These reference numbers will be included in the respective section for the XIOS module, further on in this specification.

The call has the following format:

RESOLVE_REFERENCE (call number 99H)

Function: To return the address of the requested global reference

Entry Parameters: Register C = 99H

Register E = XIOS Module ID
where XIOS Module ID is one of:
- 00 - Basic BDOS and BIOS
- 01 - Basic BDOS, Extended BDOS, and BIOS
- 13 - Multi-Window Screen Driver
- 14 - 80 Column Screen Driver
- FF - IOS Kernel

Register D = Reference Number as defined for each respective XIOS Module. This number has the range from 00 to FFH.

Exit Parameters: Register A = Status
where Status is one of:
- 00 - Search was successful with the address being returned in Register HL
- -1 - XIOS Module was not found and no address is being returned
- -2 - Reference number was not found and no address is being returned

Register HL = Address of the global reference

## 5.3 DISK SYSTEM

### 5.3.1 Introduction

The floppy disk units are attached to the Nabu PC via an interface card. The interface card is capable of supporting up to two disk drives. Each drive can be single or double density, single or double sided, full height or half height, 48 or 96 tpi. The disk drive currently provided is a single sided double density half height unit with 48 tpi.

New diskettes must be formatted to a recognizable format. The Nabu standard format is 40 tracks per side, soft sectored with 5 sectors per track and 1024 bytes per sector. The software is able to read single or double density disks produced by CP/M systems on Xerox 820, Cromemco, Osborne, {Kaypro} or IBM PC's.

PROGRAM RESPONSIBILTY

Storage of retrieval of data files are the responsibility of the individual application programs. Creation or modification of files must be handled, as well as intercepting and interpreting error codes from the file subsystem.

The only independant responsibility the end user has is in disk maintenance, i.e. format, backup, copy etc etc.. This responsibility is handled by the disk utility application programs as described in the disk utility manuals.

FILES AND DIRECTORY

The files stored on disk are CP/M version 3.0 files, stored in a CP/M directory and all calls to the directory and file handling routines are standard CP/M. The routines to do file management are supplied by Digital Research Inc. and are normally referred to as the BDOS. The BDOS interfaces to low level disk access routines called the BIOS. Application routines should do all disk access via the proper BDOS calls.

The Console Command Processor usually a part of the CP/M operating system, does not exist in the cable environment and the equivalent functions are handled via other routines.

DISK ERROR HANDLING

Errors detected by the BIOS or BDOS will be returned to the calling program, rather than resulting in an error on the users console. Application programs need to test the appropriate status on return from a BDOS call.

WARNING

The disk routines use the same buffer area in IOS as the segment handler. Therefore, before accessing the segment handling routines in IOS while you have open disk files, it is strongly recommended that you close all files, perform the segment load(s) then reset the disk system.

Programmers should reference the CP/M documentation directly about the CP/M disk features and programming requirements. In particular, the CP/M Plus User's guide - gives an overview of the organization and access of CP/M files. The CP/M plus programmer's guide gives detailed descriptions, especially sections:

2.1 Calling Conventions
2.3 BDOS File System
3. BDOS calls (refer only to file access calls)

Programmers should be aware that the disk files are handled by the DRI supplied routines, and that any other CP/M features have been implemented by Nabu in a compatible form. Section 4.2 of this guide deals with CP/M compatible calls, and contains a list of all calls.

**CP/M Version 3.0**

Version 3.0 of CP/M has several enhancements that will be of value to programmers. The extensions included in the disk support are the following:

Time and date stamping on files - refer to section 2.7.2 of the CP/M user's guide and section 2.3.8 of the CP/M programmer's guide.

Automatic diskette login - refer to section 2.3.11 of the CP/M programmer's guide.

End of file marking - refer to section 2.3.12 of the CP/M programmer's guide.

Error trapping and return to program - see section 2.3.13 of the CP/M programmer's guide.

Maximum file size is now 32Mb per file.

The application programmer needs to set up only a single control block (File Control Block - FCB) to access a file. Refer to section 2.3.3 of the CP/M programmer's guide.

## 5.4 MULTI-WINDOW SCREEN DRIVER

### 5.4.1 INTRODUCTION

This XIOS module will contain a complete set of routines which form the multi-window screen driver. These routines were formerly BOS routines contained within the IOS Kernel.

### 5.4.2 OPERATIONAL REQUIREMENTS

This XIOS module will not require any other XIOS module in order for it to function. It does however use BOS calls from within the IOS Kernel to interface with the video hardware.

### 5.4.3 MODULE SPECIFIC ERROR CODES

This XIOS module will not return any error codes specific to itself, when it has been loaded, and when the module has finished initialization or de-initialization.

### 5.4.4 MODULE INITIALIZATION

When this XIOS module is loaded, its initialization procedure is executed. This procedure will do the following:

1. Link into the IOS Kernel BOS routines as required.
2. Disable the previous screen driver.
3. Set the video screen to text mode.
4. Fill the video screen with a blue background and a blue foreground.
5. Create window #1 with size 38 columns by 24 rows; the cursor will be a flashing underline character.
6. Enable the video hardware to output to screen.
7. Enable the cursor to flash.

Windows 2, 3, 4, and 5 will be undefined after initialization.

### 5.4.4 MODULE DE-INITIALIZATION

Prior to the module being physically removed from memory, a "shut-down" or de-initialization procedure is executed. This procedure will do the following:

1. Clear the screen by filling with blanks.
2. Restore the Kernel routines such as the clock interrupt handler, to their "prior to XIOS module" state.

This procedure can not and will not restore the total context of the screen prior to the XIOS module being loaded.

### 5.4.5 DOS CALL INTERFACE

This module will be capable of decoding and executing four DOS calls. The call numbers decoded are:

8F -- DEFINE WINDOW
99 -- RETURN GLOBAL ADDRESS OF BOS ROUTINE
A2 -- INPUT STATUS FROM VIDEO SCREEN WINDOW
A3 -- OUTPUT DATA TO VIDEO SCREEN WINDOW

DEFINING VIDEO SCREEN WINDOWS

Up to five windows may be defined. Upon initialization window 1 is set up to be the full text screen. Windows may be altered or removed with the following call:

DEFINE_WINDOW (call number 8FH)

Function: Used to define a screen window for use by the VIDEO_SCREEN calls below

Entry Parameters: Register C = 8F Hex
Register DE = Pointer to WINDOW_DEFINITION_BLOCK

Exit Parameters: Register HL = Pointer to old WINDOW_CONTROL_BLOCK
or
zero if no old WCB exists

Cautions: This routine is not re-entrant.

The window is defined via the following two data structures:

```
WINDOW_DEFINITION_BLOCK:
                   DEVICE_LOCATION: BYTE;
                   WCB_POINTER:     ADDRESS;
```

Where:

DEVICE_LOCATION contains the single byte number of the window being defined. It has a range of 1 to 5.

WCB_POINTER contains a two byte pointer to a valid window control block. If this value is zero, the window becomes undefined, and thus the window is closed.

```
WINDOW_CONTROL_BLOCK:
                TOP_LEFT_ADDRESS: WORD;
                COLUMN_WIDTH:     BYTE;
                ROW_DEPTH:        BYTE;
                CURSOR_TYPE:      BYTE;
                CURSOR_PATTERN:   BYTE;
                CURSOR_X_POS:     BYTE;
                CURSOR_Y_POS:     BYTE;
                TAB_MAP:          ARRAY[1..39] OF
                                              BOOLEAN;
```

Where:

TOP_LEFT_ADDRESS contains a two byte value. This value is computed as follows:

TOP_LEFT_ADDRESS = row number * 40 + column number

Where: the row number and column number represent the top left corner of the window

This value has a range of 0 to 959 decimal

COLUMN_WIDTH contains a one byte value. It is the number of columns the window is wide. It has a range of 1 to 40.

ROW_DEPTH contains a one byte value. It is the number of rows the window is deep. It has a range of 1 to 24.

CURSOR_TYPE contains a one byte value. Two bits are defined as follows:
- bit 0 set indicates a visible cursor exists
- bit 0 clear indicates no visible cursor exists
- bit 7 set indicates the cursor is to flash
- bit 7 clear indicates the cursor is to be steady

PATTERN_NAME contains a one byte value. It is the ASCII character which is to be the cursor shape. The default window uses the underline character.

CURSOR_X_POS contains a one byte value. It is the relative cursor column position within the window. It has a range of 0 to COLUMN_WIDTH-1. It is usually set to 0.

CURSOR_Y_POS contains a one byte value. It is the relative cursor row position within the window. It has a range of 0 to ROW_DEPTH-1. It is usually set to 0.

TAB_MAP contains an array of 40 bits (5 bytes). These bits identify tab stops. If a bit is set, then a tab stop exists at that relative column number in the window.

DEFINE_WINDOW initializes one of the five windows (1 to 5) which are associated with the Video Display Physical Devices 1 to 5. If a window is already associated with the device location being set up, then the existing window is closed and a pointer to the closed WINDOW_CONTROL_BLOCK is returned in the HL Register. Otherwise 0000 is returned in the HL Register. It should be noted that windows must not be re-defined in both foreground and background tasks at the same time because the routine is not re-entrant.

## RETURN GLOBAL ADDRESS OF BOS ROUTINE

The following BOS routines have globally known entry points:

| Name | Description | Reference No. |
|---|---|---|
| WINDO | Open window | 1 |
| CLOSEW | Close window | 2 |
| SETCU | Set cursor parameters | 3 |
| GOTOX | Move cursor in window | 4 |
| PUTCH | Put character into window | 5 |
| UPSCR | Scroll window up one row | 6 |
| DOWNS | Scroll window down one row | 7 |
| LEFTS | Scroll window left one column | 8 |
| RIGHT | Scroll window right one column | 9 |
| FILLA | Fill area of window | 10 |
| DUMBT | Use window as dumb terminal | 11 |

See section 5.2.4 for complete details on using DOS call 99H.

## INPUT STATUS FROM VIDEO SCREEN WINDOW

In keeping with the standard for physical device drivers, two entry points are provided for the Video Screen Device Drivers. The first of these is as follows:

VIDEO_SCREEN: DEVICE_READY (call number A2H)

Function: Returns a data ready indication for a specified window

Entry Parameters: Register C = A2 Hex
Register E = Window Number
Where:
Window Number has a range of 1 to 5.

Exit Parameters: Register A = Return Code
Where:
Return Code = 0 indicates that the window is undefined.
Return Code = non-zero indicates that the window is defined and ready to accept data.

## OUTPUT DATA TO VIDEO SCREEN WINDOW

The second of the screen drivers has the following format:

VIDEO_SCREEN: SEND_DATA (call number A3H)

Function: Writes a character to the specified window

Entry Parameters: Register C = A3 Hex
Register E = Window Number
Where:
Window Number has a range of 1 to 5.
Register D = ASCII Character to be sent to video screen

Exit Parameters: Register A = Return Code
Where:
Return Code = 0 indicates that the window is undefined and data was not sent.
Return Code = non-zero indicates that the window is defined and data was sent.

For a list of the control characters which are accepted by this driver, see the section on BOS call DUMBT.

## 5.4.7 BOS CALL INTERFACE

This XIOS module contains eleven low-level BOS routines for using windows. Linkage to these routines is direct with their addresses being resolved with DOS call 99H as described in section 3.5.5.6.2

### OPEN A WINDOW

ROUTINE NAME: WINDO

GLOBAL REFERENCE NUMBER: 1

FUNCTION: Open a window

ENTRY PARAMETERS: REGISTER BC = Pointer to a valid WINDOW_CONTROL_BLOCK

```
Where:
     WINDOW_CONTROL_BLOCK contains:
          TOP_LEFT_ADDRESS: WORD;
          COLUMN_WIDTH:     BYTE;
          ROW_DEPTH:        BYTE;
          CURSOR_TYPE:      BYTE;
          CURSOR_PATTERN:   BYTE;
          CURSOR_X_POS:     BYTE;
          CURSOR_Y_POS:     BYTE;
          TAB_MAP:          ARRAY[1..39] OF
                                  BOOLEAN;
Where:
     TOP_LEFT_ADDRESS contains a two byte value.
          This value is computed as follows:
               TOP_LEFT_ADDRESS = row number * 40
                                  + column number
               Where: the row number and column
                      number represent the top
                      left corner of the window
          This value has a range of 0 to 959
          decimal
     COLUMN_WIDTH contains a one byte value. It
          is the number of columns the window is
          wide. It has a range of 1 to 40.
     ROW_DEPTH contains a one byte value. It is
          the number of rows the window is deep.
          It has a range of 1 to 24.
```

CURSOR_TYPE contains a one byte value. Two bits are defined as follows:
- bit 0 set indicates a visible cursor exists
- bit 0 clear indicates no visible cursor exists
- bit 7 set indicates the cursor is to flash
- bit 7 clear indicates the cursor is to be steady

PATTERN_NAME contains a one byte value. It is the ASCII character which is to be the cursor shape. The default window uses the underline character.

CURSOR_X_POS contains a one byte value. It is the relative cursor column position within the window. It has a range of 0 to COLUMN_WIDTH-1. It is usually set to 0.

CURSOR_Y_POS contains a one byte value. It is the relative cursor row position within the window. It has a range of 0 to ROW_DEPTH-1. It is usually set to 0.

TAB_MAP contains an array of 40 bits (5 bytes). These bits identify tab stops. If a bit is set, then a tab stop exists at that relative column number in the window.

EXIT PARAMETERS: REGISTER A = RETURN CODE
Where:
- RETURN CODE = 255 indicates that the Open was successful
- RETURN CODE = 0 indicates that the Open failed due to it being the sixth window or window control block not specified correctly

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
6+ Bytes of stack used

This routine is used to open a window on the screen and initialize a cursor in the window. It is passed a properly set up WINDOW_CONTROL_BLOCK. A maximum of 5 windows may be defined concurrently. If the WINDOW_CONTROL_BLOCK is not set up correctly or the 6th window is to be opened, the return code is zero.

CLOSE WINDOW

ROUTINE NAME: CLOSEW

GLOBAL REFERENCE NUMBER: 2

FUNCTION: Close an opened window

ENTRY PARAMETERS: REGISTER BC = Pointer to WINDOW_CONTROL_BLOCK to be closed
If BC = 0 then all previously opened windows are closed

EXIT PARAMETERS: REGISTER A = RETURN CODE
Where:
RETURN CODE = 0 indicates that WCB was not found in table of open windows
RETURN CODE = 255 indicates that the window was successfully closed

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
4+ Bytes of stack used

This routine is used to close a cursor window when it is no longer needed. A pointer to the WINDOW_CONTROL_BLOCK is passed in register BC. The WCB is removed from the list of active windows, and the window cursor is turned off.

SET CURSOR PARAMETERS

ROUTINE NAME: SETCU

GLOBAL REFERENCE NUMBER: 3

FUNCTION: Set the cursor parameters

ENTRY PARAMETERS: REGISTER BC = Pointer to a working Window Control Block
REGISTER D = CURSOR_TYPE
REGISTER E = PATTERN_NAME

Where:

CURSOR_TYPE contains a one byte value. Two bits are defined as follows:

- bit 0 set indicates a visible cursor exists
- bit 0 clear indicates no visible cursor exists
- bit 7 set indicates the cursor is to flash
- bit 7 clear indicates the cursor is to be steady

PATTERN_NAME contains a one byte value. It is the ASCII character which is to be the cursor shape. The default window uses the underline character.

EXIT PARAMETERS: REGISTER A = RETURN CODE
Where:
RETURN CODE = 0 indicates that WCB was not found in table of open windows
RETURN CODE = 255 indicates that the change occurred.

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
6+ Bytes of stack used

This routine is used to alter the parameters of a cursor in a cursor window which is already open. This routine MUST be used to turn off and turn on cursor flashing for windows.

MOVE CURSOR IN WINDOW

ROUTINE NAME: GOTOX

GLOBAL REFERENCE NUMBER: 4

FUNCTION: Move the cursor to new position

ENTRY PARAMETERS: REGISTER BC = Pointer to an opened Window Control Block
REGISTER D = CURSOR_X_POS
REGISTER E = CURSOR_Y_POS

Where:

CURSOR_X_POS contains a one byte value. It is the relative cursor column position within the window. It has a range of 0 to COLUMN_WIDTH-1. It is usually set to 0.

CURSOR_Y_POS contains a one byte value. It is the relative cursor row position within the window. It has a range of 0 to ROW_DEPTH-1. It is usually set to 0.

EXIT PARAMETERS: REGISTER A = RETURN CODE

Where:

*RETURN CODE = 0 indicates that* the reposition failed due to an incorrectly specified window.

RETURN CODE = 255 indicates that the reposition occurred.

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
4+ Bytes of stack used

This routine is used to re-position the cursor in a window. If the cursor is positioned outside the window the return code will indicate failure.

PUT CHARACTER IN WINDOW

ROUTINE NAME: PUTCH

*GLOBAL REFERENCE NUMBER: 5*

FUNCTION: Put an ASCII character in the window

ENTRY PARAMETERS: REGISTER BC = Pointer to an opened Window control block
REGISTER E = ASCII character with range 20H to 7EH

EXIT PARAMETERS: REGISTER A = RETURN CODE
Where:
RETURN CODE = 0 indicates that write to the window failed due to the window not being opened or the Window Control Block being incorrectly specified.
RETURN CODE = 255 indicates that the write to the window was successful.

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
6+ Bytes of stack used

This routine will output a single character at the current cursor position and advance the cursor. No control characters are interpreted, any data passed to the routine is assumed to be a character. The cursor is advanced according to the WRAP Algorithm as follows:

```
CURSOR.XPOS:=CURSOR.XPOS+1;
IF CURSOR.XPOS > (WINDOW.WIDTH - 1) THEN
    BEGIN
         CURSOR.XPOS:=0;
         CURSOR.YPOS:=CURSOR.YPOS+1;
    END
ELSE IF CURSOR.XPOS < 0 THEN
    BEGIN
         CURSOR.XPOS:=WINDOW.WIDTH-1;
         CURSOR.YPOS:=CURSOR.YPOS-1;
    END
IF CURSOR.YPOS > (WINDOW.ROWDEPTH-1) THEN
    BEGIN
         CURSOR.YPOS:=0;
         EXIT(WRAP_ALGORITHM);
    END;
IF CURSOR.YPOS < 0 THEN
    BEGIN
         CURSOR.YPOS:=WINDOW.ROWDEPTH-1;
         EXIT(WRAP_ALGORITHM;
    END;
EXIT(WRAP_ALGORITHM);
```

SCROLL WINDOW UP ONE ROW

ROUTINE NAME: UPSCR

GLOBAL REFERENCE NUMBER: 6

FUNCTION: Scroll the window up one line or row

```
ENTRY PARAMETERS:   REGISTER BC = Pointer to a complete
                       or partial WINDOW_CONTROL_BLOCK
    Where:
         WINDOW_CONTROL_BLOCK   must   contain   valid
                              values for the following:
               TOP_LEFT_ADDRESS: WORD;
               COLUMN_WIDTH:     BYTE;
               ROW_DEPTH:        BYTE;
    Where:
         TOP_LEFT_ADDRESS  contains a two byte  value.
              This value is computed as follows:
                   TOP_LEFT_ADDRESS = row number * 40
                                      + column number
                   Where: the  row  number and  column
                          number  represent  the  top
                          left corner of the window
              This  value  has  a range of  0  to  959
              decimal
         COLUMN_WIDTH contains a one byte  value.   It
              is  the number of columns the window  is
              wide.  It has a range of 1 to 40.
         ROW_DEPTH contains a one byte value.   It  is
              the  number of rows the window is  deep.
              It has a range of 1 to 24.

EXIT PARAMETERS:  REGISTER A = RETURN CODE
                   Where:
                        RETURN  CODE  = 255  indicates
                             that   the   scroll   was
                             successful
                        RETURN CODE = 0 indicates that
                             the  scroll failed due to
                             window control block  not
                             specified correctly
```

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
8+ Bytes of stack used

This routine will scroll a window up one row and replace the bottom row with blanks. Note that the area being scrolled need not be an open window. A partial WINDOW_CONTROL_BLOCK may be used to define the area to be scrolled.

SCROLL WINDOW DOWN ONE ROW

ROUTINE NAME: DOWNS

GLOBAL REFERENCE NUMBER: 7

FUNCTION: Scroll the window down one line or row

ENTRY PARAMETERS: REGISTER BC = Pointer to a complete or partial WINDOW_CONTROL_BLOCK

Where:

WINDOW_CONTROL_BLOCK must contain valid values for the following:

```
TOP_LEFT_ADDRESS: WORD;
COLUMN_WIDTH:     BYTE;
ROW_DEPTH:        BYTE;
```

Where:

TOP_LEFT_ADDRESS contains a two byte value. This value is computed as follows:

TOP_LEFT_ADDRESS = row number * 40 + column number

Where: the row number and column number represent the top left corner of the window

This value has a range of 0 to 959 decimal

COLUMN_WIDTH contains a one byte value. It is the number of columns the window is wide. It has a range of 1 to 40.

ROW_DEPTH contains a one byte value. It is the number of rows the window is deep. It has a range of 1 to 24.

EXIT PARAMETERS: REGISTER A = RETURN CODE
Where:
RETURN CODE = 255 indicates that the scroll was successful
RETURN CODE = 0 indicates that the scroll failed due to window control block not specified correctly

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
8+ Bytes of stack used

This routine will scroll a window down one row and replace the top row with blanks. Note that the area being scrolled need not be an open window. A partial WINDOW_CONTROL_BLOCK may be used to define the area to be scrolled.

SCROLL WINDOW LEFT ONE COLUMN

ROUTINE NAME: LEFTS

GLOBAL REFERENCE NUMBER: 8

FUNCTION: Scroll the window left one column

ENTRY PARAMETERS: REGISTER BC = Pointer to a complete or partial WINDOW_CONTROL_BLOCK
Where:
WINDOW_CONTROL_BLOCK must contain valid values for the following:

```
TOP_LEFT_ADDRESS: WORD;
COLUMN_WIDTH:     BYTE;
ROW_DEPTH:        BYTE;
```

Where:
TOP_LEFT_ADDRESS contains a two byte value. This value is computed as follows:
TOP_LEFT_ADDRESS = row number * 40 + column number
Where: the row number and column number represent the top left corner of the window
This value has a range of 0 to 959 decimal

COLUMN_WIDTH contains a one byte value. It is the number of columns the window is wide. It has a range of 1 to 40.
ROW_DEPTH contains a one byte value. It is the number of rows the window is deep. It has a range of 1 to 24.

EXIT PARAMETERS: REGISTER A = RETURN CODE
Where:
RETURN CODE = 255 indicates that the scroll was successful
RETURN CODE = 0 indicates that the scroll failed due to window control block not specified correctly

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
8+ Bytes of stack used

This routine will scroll a window left one column and replace the last column with blanks. Note that the area being scrolled need not be an open window. A partial WINDOW_CONTROL_BLOCK may be used to define the area to be scrolled.

SCROLL WINDOW RIGHT ONE COLUMN

ROUTINE NAME: RIGHT

GLOBAL REFERENCE NUMBER: 9

FUNCTION: Scroll the window right one column

ENTRY PARAMETERS: REGISTER BC = Pointer to a complete or partial WINDOW_CONTROL_BLOCK
Where:
WINDOW_CONTROL_BLOCK must contain valid values for the following:

```
TOP_LEFT_ADDRESS: WORD;
COLUMN_WIDTH:     BYTE;
ROW_DEPTH:        BYTE;
```

Where:
TOP_LEFT_ADDRESS contains a two byte value. This value is computed as follows:
TOP_LEFT_ADDRESS = row number * 40 + column number
Where: the row number and column number represent the top left corner of the window
This value has a range of 0 to 959 decimal
COLUMN_WIDTH contains a one byte value. It is the number of columns the window is wide. It has a range of 1 to 40.
ROW_DEPTH contains a one byte value. It is the number of rows the window is deep. It has a range of 1 to 24.

EXIT PARAMETERS: REGISTER A = RETURN CODE
Where:
RETURN CODE = 255 indicates that the scroll was successful
RETURN CODE = 0 indicates that the scroll failed due to window control block not specified correctly

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
8+ Bytes of stack used

This routine will scroll a window right one column and replace the first column with blanks. Note that the area being scrolled need not be an open window. A partial WINDOW_CONTROL_BLOCK may be used to define the area to be scrolled.

FILL AREA OF WINDOW

ROUTINE NAME: FILLA

GLOBAL REFERENCE NUMBER: 10

FUNCTION: Fill the entire area of a window

ENTRY PARAMETERS: REGISTER E = ASCII character with range 20H to 7EH
REGISTER BC = Pointer to a complete or partial WINDOW_CONTROL_BLOCK

Where:
WINDOW_CONTROL_BLOCK must contain valid values for the following:
TOP_LEFT_ADDRESS: WORD;
COLUMN_WIDTH: BYTE;
ROW_DEPTH: BYTE;

Where:
TOP_LEFT_ADDRESS contains a two byte value. This value is computed as follows:
TOP_LEFT_ADDRESS = row number * 40 + column number
Where: the row number and column number represent the top left corner of the window
This value has a range of 0 to 959 decimal
COLUMN_WIDTH contains a one byte value. It is the number of columns the window is wide. It has a range of 1 to 40.
ROW_DEPTH contains a one byte value. It is the number of rows the window is deep. It has a range of 1 to 24.

EXIT PARAMETERS: REGISTER A = RETURN CODE
Where:
RETURN CODE = 255 indicates that the fill was successful
RETURN CODE = 0 indicates that the fill failed due to window control block not specified correctly

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
8+ Bytes of stack used

This routine will fill a rectangular area on the screen with a particular character. The area being filled need not be an open window. A partial WINDOW_CONTROL_BLOCK may be used to specify the area to be filled.

## USE WINDOW AS DUMB TERMINAL

ROUTINE NAME: DUMBT

GLOBAL REFERENCE NUMBER: 11

FUNCTION: Use a window as a dumb terminal or glass teletype

ENTRY PARAMETERS: REGISTER E = ASCII character with range 0 to 7FH
REGISTER BC = Pointer to a complete WINDOW_CONTROL_BLOCK

EXIT PARAMETERS: REGISTER A = RETURN CODE
Where:
RETURN CODE = 255 indicates that the write was successful
RETURN CODE = 0 indicates that the window is not open.

CAUTIONS: This routine is not re-entrant

REGISTERS USED: A,B,C,D,E,F,HL,IX
6+ Bytes of stack used

This routine allows an opened window to be used as if it were an ASCII terminal. It will handle control characters: carriage return, line feed, delete, backspace, form feed, and horizontal tabs. The routine puts the character at the current cursor position of an opened window. It will interpret the control characters as follows:

LINE FEED: CONTROL J
If the cursor is on the bottom line of the window, the window will scroll up one line and leave the bottom line filled with SPACES and the cursor will drop straight down into this blank line. If the cursor is in the middle of the window, the cursor just drops down one line.

CARRIAGE RETURN: CONTROL M
The cursor will move to the first position of the current line.

BACKSPACE: CONTROL H
The cursor moves back one position. If the cursor is in the top-left position of the window, nothing happens.

DELETE: 7FH
The cursor backspaces one character and places a SPACE over the character.

FORM FEED: CONTROL L
The cursor is reset to the top-left position of the window and the window is filled with SPACES.

HORIZONTAL TAB: CONTROL I
The cursor is moved over to the next tab position of the current line. If no tab position is found, the cursor is placed at the start of the next line.

BELL: CONTROL G
A short tone will sound.

VERTICAL TAB: CONTROL K
The cursor moves up one line. If the cursor is on the top-most line, nothing will happen.

HOME: CONTROL ^
The cursor is reset to the top-left position of the window.

OTHER CONTROL CHARACTERS:
Nothing will happen.

## 5.5 80 COLUMN SCREEN DRIVER

### 5.5.1 INTRODUCTION

This XIOS module will contain a complete set of routines which form the 80 column screen driver. This screen driver will emulate a Lear Seigler ADM-3A type terminal on a 36 column visual video screen. A list of the control character implemented is specified in a later section.

### 5.5.2 OPERATIONAL REQUIREMENTS

This XIOS module will not require any other XIOS module in order for it to function. It does however use BOS calls from within the IOS Kernel to interface with the video hardware.

### 5.5.3 MODULE SPECIFIC ERROR CODES

This XIOS module will not return any error codes specific to itself, when it has been loaded, and when the module has finished initialization or de-initialization.

### 5.5.4 MODULE INITIALIZATION

When this XIOS module is loaded, its initialization procedure is executed. This procedure will do the following:

1. Link into the IOS Kernel BOS routines as required.
2. Disable the previous screen driver.
3. Set the video screen to text mode.
4. Fill the video screen with a blue background and a blue foreground.
5. Create a virtual screen with size 80 columns by 24 rows; the cursor will be a flashing underline character.
6. Create a visual "window" with size 36 columns by 24 rows.
7. Enable the video hardware to output to screen.
8. Enable the cursor to flash.

### 5.5.5 MODULE DE-INITIALIZATION

Prior to the module being physically removed from memory, a "shut-down" or de-initialization procedure is executed. This procedure will do the following:

1. Clear the screen by filling with blanks.
2. Restore the Kernel routines such as the clock interrupt handler, to their "prior to XIOS module" state.

This procedure can not and will not restore the total context of the screen prior to the XIOS module being loaded.

### 5.5.6 DOS CALL INTERFACE

This module will be capable of decoding and executing two DOS calls. The call numbers decoded are:

A2 -- INPUT STATUS FROM VIDEO SCREEN
A3 -- OUTPUT DATA TO VIDEO SCREEN

#### 5.5.6.1 INPUT STATUS FROM VIDEO SCREEN WINDOW

In keeping with the standard for physical device drivers, two entry points are provided for the Video Screen Device Drivers. The first of these is as follows:

VIDEO_SCREEN: DEVICE_READY (call number A2H)

Function: Returns a data ready indication for the screen driver

Entry Parameters: Register C = A2 Hex

Exit Parameters: Register A = Return Code
Where:
Return Code = 0 indicates that the video device is busy
Return Code = non-zero indicates that the video device is ready to accept data.

## 5.5.6.2 OUTPUT DATA TO VIDEO SCREEN WINDOW

The second of the screen drivers has the following format:

VIDEO_SCREEN: SEND_DATA (call number A3H)

Function: Writes a character to the screen driver.

Entry Parameters: Register C = A3 Hex
Register D = ASCII Character to be sent to video screen

Exit Parameters: None

The following is a list of the control characters interpreted:

BELL: CONTROL G
A short tone will sound.

BACKSPACE: CONTROL H
The cursor moves back one position. If the cursor is in the top-left position of the screen, nothing happens.

LINE FEED: CONTROL J
If the cursor is on the bottom line of the screen, the screen will scroll up one line and leave the bottom line filled with SPACES and the cursor will drop straight down into this blank line. If the cursor is in the middle of the screen, the cursor just drops down one line.

CURSOR UP: CONTROL K
The cursor moves up one line. If the cursor is on the top-most line, nothing will happen.

CURSOR RIGHT: CONTROL L
The cursor moves right one column. If the cursor is at the right most position on a line, a line feed action will occur

CARRIAGE RETURN: CONTROL M
The cursor will move to the first position of the current line.

CLEAR SCREEN: CONTROL Z
The cursor is reset to the top-left position of the screen and the screen is filled with SPACES.

HOME: CONTROL ^
The cursor is reset to the top-left position of the screen.

DELETE: 7FH
The cursor backspaces one character and places a SPACE over the character.

OTHER CONTROL CHARACTERS:
Nothing will happen.

## 5.6 CP/M COMPATIBLE LOGICAL DEVICE DRIVERS

### 5.6.1 INTRODUCTION

This XIOS module will contain a set of routines which are compatible with the logical device drivers found in CP/M 2.2. This is a subset of the CP/M 2.2 BDOS and does not include the disk oriented functions. These routines were formerly contained within the IOS Kernel.

### 5.6.2 OPERATIONAL REQUIREMENTS

This XIOS module may require other XIOS modules in order for it to function. It does use DOS calls 0A0H through 0A5H inclusively for interfacing to the screen, the keyboard, and the printer. These DOS calls will be found within the IOS KERNEL or XIOS modules. The user must ensure that the functions for DOS calls 0A0H to 0A5H exist in memory, prior to using the logical drivers.

### 5.6.3 MODULE SPECIFIC ERROR CODES

This XIOS module will not return any error codes specific to itself, when it has been loaded, and when the module has finished initialization or de-initialization.

### 5.6.4 MODULE INITIALIZATION

When this XIOS module is loaded, its initialization procedure is executed. This procedure will do the following:

1. Resolve the required global references in the IOS KERNEL.
2. Modify the jump address at location 1,2 in RAM such that it points to the second entry in the BIOS jump table (as per CP/M convention).

### 5.6.5 MODULE DE-INITIALIZATION

Prior to the module being physically removed from memory, a "shut-down" or de-initialization procedure is executed. This procedure will do the following:

1. Replace the modified jump location at 1,2 in RAM with that which was originally there.

### 5.6.6 DOS CALL INTERFACE

This module will be capable of decoding and executing ten DOS calls. The call numbers decoded are:

00 -- SYSTEM RESET
01 -- CONSOLE INPUT
02 -- CONSOLE OUTPUT
03 -- READER INPUT
04 -- TAPE OUTPUT
05 -- LIST OUTPUT
06 -- DIRECT CONSOLE I/O
09 -- PRINT STRING
10 -- READ CONSOLE BUFFER
11 -- GET CONSOLE STATUS

SYSTEM_RESET (call number 00H)
- performs same function as a jump to location 0000 Hex
- entry parameters:
  C Register: 00 Hex
- is not re-entrant

CONSOLE_INPUT (call number 01H)
- reads the next character from the logical console with echo. The call does not return until a character is ready. This call will only accept CP/M compatible ASCII characters. If the "YES" key is hit, a "Y" is returned. If the "NO" key is hit, a "N" is returned. All other key codes above 7FH are returned but not echoed to the screen.
- entry parameters:
  C Register: 01 Hex
- Returned Values:
  A Register: Character Input
- is not re-entrant

CONSOLE_OUTPUT (call number 02H)
- outputs a character to the logical console. Since the default physical console driver is DOS calls 0A2H and 0A3H, consult the specification for DOS call 0A3H for control character interpretation.
- entry parameters:
  C Register: 02 Hex
  E Register: Character to be output

READER_INPUT (call number 03H)

- -get a byte from the logical TAPE reader. Control will not return to the calling program until the character has been read. This call will only accept CP/M compatible ASCII characters. If the "YES" key is hit, a "Y" is returned. If the "NO" key is hit, a "N" is returned. All other key codes above 7FH are returned but not echoed to the screen.
- -entry parameters:
  - C Register: 03 Hex
- -returned value:
  - A Register: character read
- -is not re-entrant

PUNCH_OUTPUT (call number 04H)

- -output a byte to the logical TAPE punch. Since the default physical console driver is DOS calls 0A2H and 0A3H, consult the specification for DOS call 0A3H for control character interpretation.

- -entry parameters:
  - C Register: 04 Hex
  - E Register: character to be output

LIST_OUTPUT (call number 05H)

- -output a character to the logical list device
- -entry parameters:
  - C Register: 05 Hex
  - E Register: character to be output

DIRECT_CONSOLE_IO (call number 06H)

- -provides unadorned I/O from/to the logical console. Upon entry, the E register either contains an 0FF Hex, denoting a console input request, or a character to be output. If the input value if 0FF Hex, then the functions returns with the A register set to 00 if no character is ready at the logical console otherwise the A register is set to the character value input from the logical console. This call will only accept CP/M compatible ASCII characters. If the "YES" key is hit, a "Y" is returned. If the "NO" key is hit, a "N" is returned. Since the default physical console driver is DOS calls 0A2H and 0A3H, consult the specification for DOS call 0A3H for control character interpretation.

-entry parameters:
  C Register: 06 Hex
  E Register: FF Hex (input) or
              character to be output
-returned value:
  A Register: character of 00 Hex (input)
              nothing if output
-is not re-entrant

PRINT_STRING (call number 09H)

-print a string to the logical console from a buffer. The character string stored in memory at the location pointed to by the DE register is sent to the logical console. A '$' is used as a delimiter to end the print string. Since the default physical console driver is DOS calls 0A2H and 0A3H, consult the specification for DOS call 0A3H for control character interpretation.

-entry parameters:
  C Register: 09 Hex
  DE Register: pointer to string

READ_CONSOLE_BUFFER (call number 0AH)

-read a line of edited logical console input to a buffer. The input is stored in the memory buffer pointer to by the DE register. If the buffer overflows console input is terminated. The format of the buffer is:

```
MAX_BUF_SIZE:                BYTE;
NUMBER_OF_CHARACTERS_READ:   BYTE;
CHARACTER_BUFFER:            ARRAY[1..MAX_BUF_SIZE] BYTE;
```

The "GO" key (0D Hex) or CNTRL J (0A Hex) will terminate the input line. This call will only accept CP/M compatible ASCII characters. If the "YES" key is hit, a "Y" is returned. If the "NO" key is hit, a "N" is returned. All other key codes above 7FH are returned but not echoed to the screen.

-entry parameters:
  C Register: 0A Hex
  DE Register: Pointer to MAX_BUF_SIZE
               (MAX_BUF_SIZE must be set as well)
-returned values:
  Console Characters in Buffer
  NUMBER_OF_CHARACTERS_READ set
-is not re-entrant

GET_CONSOLE_STATUS (call number 0BH)
- check to see if character has been typed at logical console
- entry parameters:
  - C Register: 0B Hex
- returned value:
  - A Register: 00 Hex -No character ready
    FF Hex -Character is ready and waiting
- is not re-entrant

For more information on CP/M please refer to reference [10]. Also note some important information in section 1 concerning CP/M implementation and upgrading in the IOS.

THIS PAGE LEFT INTENTIONALLY BLANK

# APPENDICES

## APPENDIX A

### Definitions and Abreviations

| | |
|---|---|
| ASCII | An American standard for assigning code numbers to keyboard characters |
| BASIC | A commonly used computer language on personal computers |
| BDOS | Digital Research's Basic Diskette Operating System This forms part of CP/M |
| BIOS | Basic Input and Output Handlers |
| Boot ROM | Read Only Memory which is immediately executed after a NPC is powered up |
| BOS | Basic Operating System - Low level routines |
| CATV | Community Antenna Television System - It is now used to denote any cable television system |
| CP/M | Digital Research's Diskette Operating System - It is the abreviation for control processor and monitor |
| CSA | Canadian Standards Association |
| DOS | General abreviation for diskette operating system - however for the NPC it means Downloadable Operating System |
| HEAD-END | Refers to the central minicomputer system that broad-casts the software. |
| I/F | General abreviation for interface |
| I/O | General Abreviation for input and output |
| IOBYTE | A memory location used by CP/M to indicate what physical I/O is connected to which logical I/O |
| IOS | Internal Operating Software for the NABU Personal Computer |
| ISR | General abreviation for Interrupt Service Routine |
| LED | Light Emitting Diode used on front panel of NPC for indicating partial status of the NPC |

| | |
|---|---|
| NA | NABU Adaptor - It is the unit which interfaces the NPC to a CATV cable system which broadcasts software and data for use in a NPC. It was formerly called NNI |
| NNI | NABU Network Interface - It is the old name for NA. |
| NPC | NABU Personal Computer |
| PIXEL | The smallest addressable graphics unit on a TV screen. |
| RAM | Read and Write type Memory for computers |
| RF Modulator | That piece of electronic equipment which converts the digital signals of the head-end minicomputer into analog signals for broadcasting. |
| ROM | Read Only type Memory for computers |
| SPRITE | A single-coloured, moveable, positionable graphics entity with variable pixel definition and resolution. |
| SYM | It is a special key on the NPC keyboard which can be used to redefine the keyboard |
| TMS-9918A | The name for the video chip in the NPC |
| VDP | Video display processor - for the NPC it is the TMS-9918A |

## APPENDIX B

This is a summary of the complete set of current IOS functions. For the CP/M calls and DOS calls, the number directly preceeding each function is the value register C must contain prior to the call. For the BOS calls the number is the value that the link table must be initialized to in order to gain access to the proper routine.

### CP/M (Calls to location 0005H)

| | | |
|---|---|---|
| 00 | System Reset | Resets NPC |
| 01 | Console Input | Read data from console |
| 02 | Console Output | Type data to console |
| 03 | Reader Input | Read data from paper tape |
| 04 | Punch Output | Punch data on paper tape |
| 05 | List Output | List data to printer |
| 06 | Direct Console I/O | Unadorned console I/O |
| 09 | Print String | Print message in buffer |
| 0A | Read Console Buffer | Read message in buffer |
| 0B | Get Console Status | Return status of console |
| 0C | Get Version Number | Not Implemented |

### Downloadable Operating Software (DOS) (Calls to location 0008H)

#### Segment Routines

| | | |
|---|---|---|
| 80 | Reset Device | Reset logical device |
| 82 | Get Status | Get adaptor status |
| 83 | Set Status | Set adaptor status |
| 84 | Load Segment | Load segment from cable |
| 87 | SEG$CST Base Address | Return control status block |
| 88 | Directory Search | |
| 96 | Load XIOS Module | |
| 97 | Unload XIOS Module | |
| 99 | Resolve Global Reference | |

#### I/O Service Routines

| | | |
|---|---|---|
| 8A | I/O Router: Attach | Set phys dev to log dev |
| A0 | Human Input: Device Ready | Keyboard ready |
| A1 | Human Input: Get Data | Get keyboard data |
| A2 | Video Screen: Device Ready | Screen ready |
| A3 | Video Screen: Send Data | Send data to screen |
| A4 | Printer: Device Ready | Printer ready |
| A5 | Printer: Send Data | Send printer data |

Multitasking Routines

| | | |
|---|---|---|
| 8B | Clock User: Task Attach | Attach task to system clk |
| 8C | Clock User: Task Remove | Remove task |
| 8D | Device User: Task Attach | Attach device to clk |
| 8E | Device User: Task Remove | Remove device from clk |

Miscellaneous

| | | |
|---|---|---|
| 90 | Link BOS Routines | Set up linktable for BOS |
| 91 | Set SYM key Table | Redefinition table for SYM |
| 92 | Read Real Time Clock | |
| 93 | Set Real Time Clock | |
| 94 | Configuration | Return system configuration |

Basic Operating Software (BOS) (Called via link table)

Video Routines

| | | |
|---|---|---|
| 00 | VREGRD | Reads TMS-9918A video display register |
| 01 | VTABRD | Reads current table base address ptrs |
| 04 | VREGWR | Writes video display register |
| 05 | VSTATRD | Reads video status register |
| 06 | VNAMEST | Sets the pattern name address |
| 07 | VCOLRST | Sets the colour table address |
| 08 | VPTRNST | Sets the pattern table address |
| 09 | VATRIST | Sets the sprite attributes table addr |
| 0A | VSPRIST | Sets the sprite table address |
| 0B | VBLKON | Blanks the video display |
| 0C | VBLKOFF | Turns on the video display |
| 0D | VRAMRD | Reads a single byte of VRAM |
| 0E | VRAMWR | Writes a single byte of VRAM |
| 0F | FASTL8 | Write a string (256 max) of bytes to VRAM |
| 10 | FASTLD | Write a string (16 K max) of bytes to VRAM |
| 11 | FASTD8 | Read string of bytes (256 max) from VRAM |
| 12 | FASTDU | Read string of bytes (16 K max) from VRAM |
| 13 | VRAML8 | Same as FASTL8 but interrupt protected |
| 14 | VRAMLD | Same as FASTLD but interrupt protected |
| 15 | VRAMD8 | Same as FASTD8 but interrupt protected |
| 16 | VRAMDU | Same as FASTDU but interrupt protected |
| 17 | SPMARK | Mark end of a sprite attributes table |
| 18 | SPMOVE | Move a sprite on the video screen |
| 19 | SPCOLR | Set the colour of a sprite |
| 1A | SPNAME | Set pattern name assoc. with a sprite |
| 1B | RPATRN | Load pattern def'ns into Screen table |
| 1C | LPATRN | Load pattern def'ns into VRAM |
| 1D | CHADR | Return VRAM addr for a certain pattern |
| 1E | VFILL | Fill block of VRAM with a character |
| 1F | XYLOC | Return name tab addr for any XY loc |

| | | |
|---|---|---|
| 20 | PUTPAT | Put pattern at any XY loc |
| 21 | GETPAT | Get pattern from any XY loc |
| 22 | SETMSG | Set up screen message |
| 23 | PUTMSG | Put a message on screen |
| 24 | GETMSG | Get a message from screen |
| 25 | VSETTXT | Set video for text mode |
| 26 | VSETG1 | Set video for Graphics 1 mode |
| 27 | VSETG2 | Set video for Graphics 2 mode |
| 28 | VSETSPA | Set sprite size and magnification |
| 3A | VMOVI | Move data in VRAM up quickly |
| 3B | VMOVD | Move data in VRAM down quickly |
| 3C | FASTRD | Unprotected single byte VRAM read |
| 3D | FASTWR | Unprotected single byte VRAM write |

Audio Routines

| | | |
|---|---|---|
| 35 | AUDRD | Read audio chip register |
| 36 | AUDWR | Write the audio chip register |

Miscellaneous BOS Routines

| | | |
|---|---|---|
| 02 | CRBEG | Start of a critical region |
| 03 | CREND | End of a critical region |
| 29 | MUL88 | Multiply two eight bit values |
| 37 | CLKPRM | Control real time processing |
| 38 | HOINIT | Initialize IOS |
| 39 | CREGWR | Write to the control port |
| 3E | SETMSK | Write hardware interrupt control register |

## APPENDIX C

The following is a sample program to demonstrate the use of the video display processsor and the audio generator. This program assumes that the M80 assembler (copyright Microsoft) is used.

The program will place messages on the screen, move a red circular or a blue square sprite around on the screen under the control of a joystick and a clock attach routine.

```
;***************************************************************************
;
;
;        PROGRAM NAME: DEMO.MAC
;
;------------------------------------------
;
;
;        DESCRIPTION: DEMONSTRATION PROGRAM TO INTRODUCE
;                     THE 9918A VIDEO DISPLAY PROCESSOR, AUDIO
;                     GENERATOR and the IOS.
;
;
;***************************************************************************

            .Z80
            .RADIX 10        ;USE BASE 10

            ;EXTERNAL FUNCTIONS
            ;THESE LABELS REFERENCE CODE OUTSIDE OF THE MAIN PROGRAM.

            EXTRN TCHAR      ;PATTERN DEFINITIONS
            EXTRN SPRPAT     ;SPRITE PATTERN DEFINITIONS

;EQUATES

BLACK     EQU       01
MGREEN    EQU       02
WHITE     EQU       0FH
DBLUE     EQU       04
;****************************************************************
```

```
;MACRO DEFINITIONS

PCALL       MACRO       SUBR, PARM1, PARM2, PARM3
            IFNB        <PARM1>
            LD          BC, PARM1
            ENDIF
            IFNB        <PARM2>
            LD          DE, PARM2
            ENDIF
            IFNB        <PARM3>
            LD          HL, PARM3
            ENDIF
            CALL        SUBR
            ENDM
;
DEFMSG      MACRO       XPOS, YPOS, MSG
            LOCAL       END, START
            DB          XPOS
            DB          YPOS
            DB          END-START
START:      DB          MSG
END:
            ENDM
;
SETCOLR     MACRO       BACK,TEXT
            IFB         <TEXT>
             PCALL      VREGWR, 07, 10H+BACK
            ELSE
             PCALL      VREGWR, 07, TEXT*10H+BACK
            ENDIF
            ENDM
;
N.CLKAT           MACRO    TASKADR
                  LD       DE,TASKADR
                  LD       C,08BH
                  CALL     NABUSYS
                  ENDM
;
N.CLKRV           MACRO    TASKADR
                  LD       DE,TASKADR
                  LD       C,08CH
                  CALL     NABUSYS
                  ENDM
;
N.LINKIO          MACRO    IOSPTR
                  LD       DE,IOSPTR
                  LD       C,090H
                  CALL     NABUSYS
                  ENDM
```

```
;
N.DEVRDY          MACRO   DEVICE,LOCATN
                  LD      E,LOCATN
                  LD      C,DEVICE*2+0A0H
                  CALL    NABUSYS
                  ENDM
;
N.DEVIO           MACRO   DEVICE, LOCATN, DATA
                  IFNB    <DATA>
                   LD     A,DATA
                   LD     D,A
                  ENDIF
                  LD      E,LOCATN
                  LD      C,DEVICE*2+0A1H
                  CALL    NABUSYS
                  ENDM
;

;*************************************************************************
;*                            DATA AREA                                  *
;*************************************************************************

           ;TASK CONTROL BLOCK FOR END OF PROGRAM

   TSKEND::
      NEXT: DW 0           ;LINKED LIST POINTER
    ENDINT: DB 15          ;EXECUTE TASK EVERY 1/4 SEC
   ENDINIT: DB 5H          ;WAIT 5/60 SEC BEFORE EXECUTION
    ENDADR: DW 1H          ;TASK ADDRESS

           ;TASK CONTROL BLOCK FOR SPRITE MOVEMENT

   TSKMSP::
     NEXT1: DW 1           ;NEXT TASK IN LINKED LIST
     SPINT: DB 1           ;EXECUTE TASK EVERY 1/60 OF A SECOND
   SPINTIT: DB 1           ;WAIT 1/60 OF A SEC BEFORE EXECUTION
    SPRADR: DW 1           ;TASK ADDRESS

           ;DEFINE BYTES FOR VARIABLES

      X:     DB 1          ;X POSITION OF SPRITE
      Y:     DB 1          ;Y POSITION OF SPRITE
   COLRR:    DB 1          ;CURRENT COLOUR OF SPRITE 7=RED 8=CYAN
   OLDIR:    DB 1          ;OLD DIRECTION OF SPRITE
   XFLAG:    DB 1          ;SOUND ENABLE FOR VERT. MOTION 1=ENABLED
   YFLAG:    DB 1          ;SOUND ENABLE FOR HORIZ. MOTION
   CFLAG:    DB 1          ;COLOUR FLAG. PREVENTS RAPID COLOUR CHANGES
```

```
          ;DEFINE ALL THE MESSAGES TO BE PRINTED

     MSG1:  DEFMSG 9H,3,'WELCOME TO NABU '
     MSG2:  DEFMSG 3H,11,'SAMPLE PROGRAM '
     MSG3:  DEFMSG 3H,13,'PRESS C KEY TO CONTINUE '
     MSG4:  DEFMSG 3H 14,'TO JOYSTICK PORTION OF TEST '
     MSG5:  DEFMSG 6H 18,'PRESS ESC KEY TO STOP '

;************************************************************************
;*                    START OF EXECUTION                                *
;************************************************************************

START::     LD SP,(0006)          ;SET STACK POINTER AT TOP OF MEMORY
            N.LINKIO LNKTB##      ;SET UP IOS JUMP TABLE

;************************************************************************
;*                THIS BLOCK OF CODE INITIALIZES THE VIDEO              *
;*                CHIP REGISTERS, LOADS THE ASCII CHARACTER             *
;*                SET AND SETS UP THE COLOUR TABLE FOR                  *
;*                WHITE LETTERS ON A BLUE BACKGROUND.                   *
;************************************************************************

            CALL VSETG1              ;SET  GRAPHIC1 MODE
            PCALL VPTRNST,0          ;SET PATTERN TABLE ADDRESS
            PCALL VNAMEST,1C00H      ;SET PATTERN NAME TABLE  ADDRESS
            PCALL VATRIST,1F00H      ;SET SPRITE ATTRIBUTE TABLE ADDRESS
            PCALL VCOLRST,2000H      ;SET COLOUR TABLE ADDRESS
            PCALL VSPRIST,3800H      ;SET SPRITE GENERATOR TABLE ADDRESS
            SETCOLR DBLUE,WHITE      ;WHITE LETTERS ON BLUE BACKGRND

;************************************************************************
;*                DISABLE THE SOUND ON THE AUDIO REGISTER               *
;************************************************************************

            PCALL AUDIOWR,7,3FH           ;SET CONTROL REGISTER TO ZERO

            PCALL RPATRN,TCHAR            ;LOAD ASCII SET

            PCALL VRAML8,20H,CLR1,2000H  ;LOAD COLOR TABLE WHITE ON BLUE

;************************************************************************
;*                THIS BLOCK OF CODE BLANKS THE SCREEN                  *
;*                AND WRITES MESSAGES ON THE SCREEN.                    *
;************************************************************************

            PCALL VFILL,960,20H,1C00H    ;FILL VIDEO SCREEN WITH BLANKS
            CALL VBLKOFF                 ;TURN THE SCREEN ON
```

```
            ;PRINT MESSAGES TO SAY HELLO AND PROMPT FOR ESC KEY

            PCALL PUTMSG,MSG1
            PCALL PUTMSG,MSG2
            PCALL PUTMSG,MSG3
            PCALL PUTMSG,MSG4

;***************************************************************************
;*             THIS BLOCK POLLS FOR THE 'C' KEY BEFORE                     *
;*             CONTINUING. ENDD IS THEN ATTACHED TO THE                    *
;*             CLOCK ISR TO CHECK FOR 'ESC' KEY INDICATING                 *
;*             END OF DEMO.                                                *
;***************************************************************************

    LOOP:   LD E,0FFH                      ;LOOP UNTIL THE 'C' KEY IS HIT
            LD C,6
            CALL 0005
            CP 'C'
            JP NZ, LOOP

            PCALL VFILL 960,20H,1C00H
            PCALL PUTMSG,MSG5
            PCALL VRAML8,20H,CLR2,2000H    ;LOAD COLOR TABLE BLACK ON GREEN
            SETCOLR MGREEN,BLACK

            LD HL,ENDD
            LD (ENDADR),HL
            N.CLKAT TSKEND

;***************************************************************************
;*            THIS BLOCK INITIALLY SETS UP SPRITE PATTERN                  *
;*            AND INITIALLY PLACES A RED CIRCULAR SPRITE                   *
;*            ON THE SCREEN.                                               *
;***************************************************************************

            ;SET UP SPRITES
            PCALL LPATRN,SPRPAT,3800H      ;LOAD SPRITE PATTERN
            PCALL SPNAME 0,0               ;SPRITE 0,PATTERN 0
            PCALL SPMARK 1                 ;END OF SPRITE ATTRIBUTE


            LD A,6                         ;SET SPRITE TO RED
            LD (COLRR),A
            PCALL SPCOLR 0,(COLRR)
            LD A,0
            LD (OLDIR),A                   ;INITIALIZE OLDIR TO 0
```

```
            LD A,30
            LD (Y),A
            LD A,40                        ;SET INITIAL SPRITE
            LD (X),A                       ;POSITION TO 40,30

            LD A,1                         ;TO PREVENT CONTINUOUS
            LD (XFLAG),A                   ;SOUND WHILE TRAVELLING ALONG
            LD (YFLAG),A                   ;HORIZ AND VERT CENTERS, ENABLE
                                           ;FLAGS
;**************************************************************************
;*          THIS BLOCK ATTACHES SPRMOV TO THE CLOCK ISR TO HANDLE          *
;*          SPRITE MOVEMENT AND MAKING 'DING' 'DONG' SOUNDS                *
;**************************************************************************

            LD HL,SPRMOV
            LD (SPRADR),HL                 ;ATTACH SPRITE MOVE
            N.CLKAT TSKMSP                 ;TO CLOCK

   INLOOP:  JP INLOOP                      ;INFINITE LOOP


;
;**************************************************************************
;
;
;       ROUTINE NAME:SPRMOV
;
;------------------------------------------
;
;       FILE NAME:    DEMO.MAC
;
;       DESCRIPTION:USED TO DETERMINE THE NEW SPRITE POSITION, TO PRODUCE
;                   SOUND. THIS ROUTINE IS ATTACHED TO THE CLOCK.
;
;
;       PARAMETERS PASSED: none
;
;       PARAMETERS RETURNED:none
;
;       REGISTERS CLOBBERED:REGISTER SAVED BY CLOCK
;
;       GLOBALS ACCESSED:  none
;
;       GLOBALS WRITTEN:   none
;
;       COMMENTS and WARNINGS:
;
;
```

```
;*************************************************************************
;*                THIS BLOCK OF CODE DETERMINES IF THE JOYSTICK          *
;*                IS READY. IF DATA IS READY, THEN N.DEVIO               *
;*                OBTAINS THE NEW DATA IN THE ACCUMULATOR.               *
;*************************************************************************

   SPRMOV::
            LD A,1          ;RESET FLAG
            LD (CFLAG),A
            N.DEVRDY 0,02   ;CHECK IF JOYSTICK HAS DATA
            JP NZ,CONT      ;IF NEW DATA IS READY THEN GET IT
            LD A,(OLDIR)    ;ELSE USE OLD DIRECTION
            JP MOV

    CONT:   N.DEVIO 0,02    ;GET NEW DATA
            LD (OLDIR) ,A   ;SAVE THE NEW DIRECTION

;************************************************************************
;*        THIS BLOCK OF CODE DETERMINES WHAT THE NEW DIRECTION IS.      *
;*        BITS ARE SET IN THE RETURN VALUE FROM N.DEVIO ACCORDING       *
;*        TO WHAT THE JOYSTICK POSITION IS.                             *
;*                                                                      *
;*     IF BIT 0 IS SET THEN MOVE LEFT                                   *
;*     IF BIT 1 IS SET THEN MOVE DOWN                                   *
;*     IF BIT 3 IS SET THEN MOVE RIGHT                                  *
;*     IF BIT 4 IS SET THEN MOVE UP                                     *
;*     IF BIT 5 IS SET THEN CHANGE THE SPRITE CLOUR AND PATTERN         *
;************************************************************************

    MOV:    SRA A           ;SHIFT THE BITS TO THE RIGHT
            CALL C,LEFT     ;AND CALL THE APPROPRIATE
            SRA A           ;ROUTINE IF THE BIT IS SET
            CALL C,DOWN
            SRA A
            CALL C,RGHT
            SRA A
            CALL C,UPP
            SRA A
            CALL C , FIRE
```

```
;*************************************************************************
;*      THIS BLOCK OF CODE DETERMINES WHETHER THE SPRITE HAS CROSSED    *
;*      THE VERTICAL LINE. IF IT HAS MAKE THE DONG SOUND.               *
;*************************************************************************

            LD A,(X)            ;CHECK IF SPRITE CROSSES VERT LINE
            CP 115
            JP NZ ,NOSNDX       ;IF NOT THEN SKIP VERT SOUND
            LD A,(YFLAG)
            CP 0                ;IS SPRITE STILL ON VERT LINE?
            JP Z,NOSNDX         ;YES-SKIP VERT SOUND
            LD A,0
            LD (YFLAG),A        ;RESET FOR SOUND

            ;PRODUCE SOUND FOR CROSSING VERICAL LINE
            PCALL AUDIOWR 0,120         ;SET TONE
            PCALL AUDIOWR 7,62          ;ENABLE CHANNEL A
            PCALL AUDIOWR 8,31          ;MAXIMUM AMPLITUDE ,ENABLE ENV.
            PCALL AUDIOWR 12,56         ;SET UP ENVELOPE
            PCALL AUDIOWR 13,0

;*************************************************************************
;*     THIS BLOCK OF CODE DETERMINES WHETHER THE SPRITE HAS CROSSED     *
;*     THE HORIZONTAL LINE. IF IT HAS MAKE THE DING SOUND.              *
;*************************************************************************

   NOSNDX:  ;CHECK FOR HORIZ. SOUND
            LD A,(Y)
            CP 90                    ;CROSS HORIZ LINE?
            JP NZ,NOSND              ;NO-  THEN NO SOUND
            LD A,(XFLAG)
            CP 0
            JP Z,NOSND
            LD A,0
            LD (XFLAG),A             ;SET FLAG

            ;PRODUCE SOUND FOR CROSSING HORIZ. LINE
            PCALL AUDIOWR 0,32    ;SELECT TONE
            PCALL AUDIOWR 7,62    ;ENABLE CHANNEL A
            PCALL AUDIOWR 8,31    ;MAX. AMP. ENABLE ENV.
            PCALL AUDIOWR 12,56   ;SET UP ENVELOPE
            PCALL AUDIOWR 13,0


   NOSND:   PCALL SPMOVE 0,(Y),(X)  ;MOVE SPRITE ON SCREEN
            RET
```

```
;************************************************************************
;
;
;       ROUTINE NAME:LEFT
;
;-----------------------------------------
;
;       FILE NAME:   DEMO.MAC
;
;       DESCRIPTION: UPDATES THE SPRITE POSITION 1 PIXEL TO THE LEFT
;
;
;       PARAMETERS PASSED: none
;
;       PARAMETERS RETURNED:none
;
;       REGISTERS CLOBBERED:none
;
;       GLOBALS ACCESSED:  none
;
;       GLOBALS WRITTEN:   none
;
;       COMMENTS and WARNINGS:
;
;


    LEFT:    PUSH AF         ;SAVE AF REGISTER
             LD A,1
             LD (YFLAG),A    ;RESET FLAG FOR SOUND
             LD A,(X)
             DEC A           ;UPDATE X POSTION
             LD (X),A
             JP NZ,LR
             LD A ,250       ;IS SPRITE AT THE EDGE OF SCREEN
             LD (X),A
      LR:    POP AF
             RET
```

```
;*********************************************************************
;
;
;       ROUTINE NAME:DOWN
;
;-------------------------------------------
;
;       FILE NAME:   DEMO.MAC
;
;       DESCRIPTION:MOVES THE SPRITE'S POSITION 1 PIXEL DOWN
;
;       PARAMETERS PASSED: none
;
;       PARAMETERS RETURNED:none
;
;       REGISTERS CLOBBERED:none
;
;       GLOBALS ACCESSED:  none
;
;       GLOBALS WRITTEN:   none
;
;       COMMENTS and WARNINGS:
;
;

  DOWN:   PUSH AF
          LD A,1
          LD (XFLAG),A
          LD A,(Y)
          INC A
          LD (Y),A
          CP 180
          JP NZ ,RD
          LD A ,0
          LD (Y),A
 RD:      POP AF
          RET


  RGHT:   PUSH AF
          LD A,1
          LD (YFLAG),A
          LD A,(X)
          INC A
          LD (X),A
          CP 245
          JP NZ ,RR
          LD A ,0
          LD (X),A
    RR:   POP AF
          RET
```

```
;**************************************************************************
;
;
;       ROUTINE NAME:UPP
;
;-----------------------------------------
;
;       FILE NAME:   DEMO.MAC
;
;       DESCRIPTION:MOVES THE SPRITE ONE PIXEL UP
;
;
;       PARAMETERS PASSED: none
;
;       PARAMETERS RETURNED:none
;
;       REGISTERS CLOBBERED:none
;
;       GLOBALS ACCESSED:  none
;
;       GLOBALS WRITTEN:   none
;
;       COMMENTS and WARNINGS:
;
;

    UPP:    PUSH AF
            LD A,1
            LD (XFLAG),A
            LD A,(Y)
            DEC A
            LD (Y),A
            JP NZ,UR
            LD A,180
            LD (Y),A
    UR:     POP AF
            RET
```

```
;*************************************************************************
;
;
;       ROUTINE NAME:FIRE
;
;-----------------------------------------
;
;       FILE NAME:   DEMO.MAC
;
;       DESCRIPTION: WHEN FIRE BUTTON IS DEPRESSED, A RED CIRCLULAR
;                    SPRITE IS TOGGLED TO A BLUE SQUARE OR BACK AGAIN.
;
;
;       PARAMETERS PASSED: none
;
;       PARAMETERS RETURNED:none
;
;       REGISTERS CLOBBERED:none
;
;       GLOBALS ACCESSED:  none
;
;       GLOBALS WRITTEN:   none
;
;       COMMENTS and WARNINGS:
;
;

    FIRE:   PUSH AF
            LD A,(CFLAG)
            CP 0                    ;HAS SPRITE BEEN CHANGED RECENTLY
            JP Z,FIRER              ;YES -THEN RETURN
            LD A,0
            LD (CFLAG),A            ;RESET FLAG
            LD A ,(COLRR)
            CP  7                   ;IS IT  RED
            JP Z ,REDD              ;YES- CHANGE SPRITE TO A SQUARE
            INC A
            LD (COLRR),A
            PCALL SPCOLR 0,(COLRR) ;AND CHANGE THE COLOUR
            PCALL SPNAME 0,5        ;MAKE IT A CIRCLE
            JP FIRER                ;GOTO RETURN
   REDD:    PCALL SPCOLR 1,(COLRR) ;CHANGE COLOUR TO BLUE
            DEC A
            LD (COLRR),A
            PCALL SPCOLR 0,(COLRR) ;CHANGE IT TO BLUE
            PCALL SPNAME 0,1        ;MAKE THE SPRITE A SQUARE
   FIRER:   POP AF
            RET
```

```
;*************************************************************************
;
;
;       ROUTINE NAME:ENDD
;
;------------------------------------------
;
;       FILE NAME:   DEMO.MAC
;
;       DESCRIPTION:DETERMINES WHETHER THE ESC HAS BEEN DEPRESSED
;                   AND IF IT HAS REBOOT THE SYSTEM.
;
;
;       PARAMETERS PASSED: none
;
;       PARAMETERS RETURNED:none
;
;       REGISTERS CLOBBERED:none
;
;       GLOBALS ACCESSED:  none
;
;       GLOBALS WRITTEN:   none
;
;       COMMENTS and WARNINGS:
;
;

        ;POLL FOR ESCAPE KEY
ENDD:   PUSH AF
        N.DEVRDY 0,01           ;IS THE KEYBOARD READY
        CP 0
        JP Z,NOEND              ;YES-THEN GET DATA ELSE RETURN
        N.DEVIO 0,01            ;GET DATA
        CP 1BH                  ;IS IT THE ESC KEY
        JP NZ ,NOEND            ;NO -RETURN
        LD C,0                  ;YES REBOOT CPM
        JP 0
NOEND:  POP AF
        RET
```

```
;*********************************************************************
;*            THIS IS THE DATA FOR THE COLOUR TABLE                  *
;*********************************************************************

        .RADIX 16
CLR1:           DB 0F4,0F4,0F4,0F4,0F4,0F4,0F4,0F4
                DB 0F4,0F4,0F4,0F4,0F4,0F4,0F4,0F4
                DB 0F4,0F4,0F4,0F4,0F4,0F4,0F4,0F4
                DB 0F4,0F4,0F4,0F4,0F4,0F4,0F4,0F4

CLR2:
                DB 012,012,012,012,012,012,012,012
                DB 012,012,012,012,012,012,012,012 ;COLOR TABLE ENTRIES
                DB 012,012,012,012,012,012,012,012
                DB 012,012,012,012,012,012,012,012
                END

;*****************************************************************
;          The SPRPAT.MAC file
;*****************************************************************



           .Z80
           CSEG
           .RADIX 2
;
;
SPRPAT::
;
           DB 008H
;
;
 DB 000H,000H,001H,00FH,01FH,03FH,03FH,07FH,07FH
 DB 001H,07FH,03FH,03FH,01FH,00FH,003H,001H,000H
 DB 002H,000H,080H,0C0H,0F0H,0F8H,0FCH,0FCH,0FEH
 DB 003H,0FEH,0FCH,0FCH,0F8H,0F0H,0C0H,080H,000H
 DB 004H,0FFH,080H,080H,080H,080H,080H,080H,080H
 DB 005H,080H,080H,080H,080H,080H,080H,080H,0FFH
 DB 006H,0FFH,001H,001H,001H,001H,001H,001H,001H
 DB 007H,001H,001H,001H,001H,001H,001H,001H,0FFH
;
            END
;
```

```
;***************************************************************************
;
;
;       ROUTINE NAME:           LNKTB
;
;-----------------------------------------
;
;       FILE NAME:              LINKTAB.MHO
;
;       DESCRIPTION:
;               LNKTB is a driver table used by the application to establish
;               user access to IOS routines. The table must exist if
;               any of the IOS routines are to be used. Before the routines
;               may be accessed, the table must be initialized.
;
;               The table consists of all the IOS routines associated with
;               VDP, windows and cursors, and attaching tasks to the
;               clock interrupt. To use the table, delete any entries which
;               are not called by your software. This leaves only the
;               routines accessed by your code.
;
;               After the unused entries are deleted, the table must be
;               assembled and the assembled version included in the
;               final link of the application.
;
;       AUTHOR:                 Trevor Pearce
;       DATE and ISSUE:         August 4, 1982  Version 1.0
;       CATALOGUE ID:           HCF - AS - 0051
;
;       PARAMETERS PASSED:      none
;
;       PARAMETERS RETURNED:    none
;
;       REGISTERS CLOBBERED:    none
;
;       GLOBALS ACCESSED:       all VIDEO, SCREEN and CURE entry points
;
;       GLOBALS WRITTEN:        all accessed globals are written during
;                               initialization
;
;       COMMENTS and WARNINGS:
;
```

```
                .Z80
                .RADIX 10
LNKTB::
                DB              (TABEND-TABSTRT)/3 ; Do not delete this line
;
TABSTRT:                                 ; Do not delete this line
;
REGRD::
VREGRD::        DB              00H,0,0
;
VTABR::
VTABRD::        DB              01H,0,0
;
CRBEG::         DB              02H,0,0
;
CREND::         DB              03H,0,0
;
REGWR::
VREGWR::        DB              04H,0,0
;
STATR::
VSTATRD::       DB              05H,0,0
;
NAMST::
VNAMEST::       DB              06H,0,0
;
COLST::
VCOLRST::       DB              07H,0,0
;
PTRST::
VPTRNST::       DB              08H,0,0
;
ATRST::
VATRIST::       DB              09H,0,0
;
SPRST::
VSPRIST::       DB              0AH,0,0
;
BLKON::
VBLKON::        DB              0BH,0,0
;

               ........ etc. etc. etc.

;
TABEND: ; Do not delete this line
;
                END
```